UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROPOSTA DE RASTREABILIDADE ENTRE MUDANÇAS DE REQUISITOS E SEUS CASOS DE TESTES AUTOMATIZADOS

Antonio de Azevedo Donatti

Leonardo Augusto da Silva

Florianópolis - Santa Catarina, 2017

UNIVERSIDADE FEDERAL DE SANTA CATARINA

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

PROPOSTA DE RASTREABILIDADE ENTRE MUDANÇAS DE REQUISITOS E SEUS CASOS DE TESTES AUTOMATIZADOS

Antonio de Azevedo Donatti

Leonardo Augusto da Silva

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação

Florianópolis / SC

2017/2

Antonio de Azevedo Donatti

Leonardo Augusto da Silva

Proposta de rastreabilidade entre mudanças de requisitos e seus casos de testes automatizados.

Orientador:

_____

Professor Raul Sidnei Wazlawick

Banca Examinadora:

_____

Prof. Ricardo Pereira e Silva

_____

Profa. Patrícia Vilain

DEDICATÓRIA

Dedicamos o presente trabalho às nossas famílias, por toda paciência, compreensão, apoio e incentivo, e aos nossos amigos por nos desafiarem a darmos nosso melhor ao longo de toda trajetória acadêmica.

"Wars come and go, but my soldiers stay eternal"

(Tupac Amaru Shakur)

AGRADECIMENTOS

Gostaríamos de agradecer ao laboratório Bridge da Universidade Federal de Santa Catarina por todo o apoio no desenvolvimento e aplicação de nossa pesquisa. Sem os profissionais envolvidos nada teria sido possível.

RESUMO

A documentação dos requisitos é o principal artefato oriundo da engenharia de requisitos.

Contudo, inevitavelmente, esta documentação poderá sofrer mudanças que irão impactar em diversos estágios do desenvolvimento do *software*. Em um ambiente de desenvolvimento onde os testes automatizados são elaborados de acordo com os requisitos documentados, tais mudanças devem ser rastreadas de forma visível e ágil, a fim de que a manutenção dos testes automatizados seja a menos custosa possível.

Sendo assim, o presente trabalho desenvolve e avalia uma ferramenta para o auxílio da rastreabilidade entre requisitos documentados e seus casos de testes automatizados.

Palavras-chave: 1. Documentação de requisitos 2. Testes Automatizados 3. Rastreabilidade de requisitos 4. Engenharia de *Software*

ABSTRACT

The requirements documentation is the main artifact provided by the requirements engineering.

Yet, inevitably, this documentation can be changed, impacting several stages of the software development. In a development environment, where automated testing is based on those documented requirements, such changes must be tracked visibly and responsibly, in order that the maintenance of automated test is the least costly possible.

Therefore, the present work develops and evaluate a tool to serve the traceability between documented requirements and their automated test cases.

Keywords: 1. Requirements documentation 2. Automated testing 3. Requirements traceability 4. Software engineering

LISTA DE REDUÇÕES

CDT - Casos de teste

Automatizador - Tester que escreve testes automatizados.

LISTA DE FIGURAS

LISTA DE GRÁFICOS

SUMÁRIO

1 INTRODUÇÃO

A deficiência no tratamento de requisitos tem sido apontada como a principal causa de fracassos de projetos de *software* [12]. Tal tratamento pode ter sido gerado devido a uma mudança no requisito.

Para um ambiente de desenvolvimento de *software* ser considerado ágil, o mesmo deve aceitar a mudança e adaptar-se a ela. O problema não é a mudança em si, mesmo porque ela ocorrerá de qualquer forma. O problema é como receber, avaliar e responder às mudanças [13].

Os testes automatizados verificam automaticamente funcionalidades do sistema e registram efeitos colaterais obtidos. Esta abordagem permite que todos os casos de teste sejam facilmente e rapidamente repetidos a qualquer momento e com pouco esforço [14].

Contudo, casos de teste automatizados que estejam associados a requisitos do *software*, também estarão sujeitos às mesmas mudanças. Como a parte de teste chega a custar até 50% do tempo de desenvolvimento, todo o tempo poupado nesta etapa é válido [15]. Sendo assim, o presente trabalho visa propor e avaliar uma ferramenta que auxilie o automatizador a identificar estas mudanças para que seus casos de teste sejam revalidados.

2 DELIMITAÇÃO DO TRABALHO

Este trabalho limita-se ao desenvolvimento e aplicação de uma ferramenta cujo objetivo visa o auxílio na identificação de casos de teste automatizados que necessitam de reavaliação. Tal reavaliação só é apontada pela ferramenta se e somente se, o caso de teste estiver vinculado à um requisito documentado, já que o rastreio é feito a partir do estado inicial do mesmo e suas futuras alterações. Caso houver adições ou exclusões de requisitos, a solução não traz resultado, uma vez que apenas requisitos identificados manualmente são rastreados. Sendo assim, tal delimitação é indicada para trabalhos futuros. Outros artefatos produzidos pela análise e engenharia de *software*, como casos de uso, dicionário de dados, modelos de arquitetura, entre outros que podem derivar casos de teste, não são abordados.

A avaliação, realizada com o intuito de verificar a usabilidade prática da ferramenta, limitou-se ao universo de automatizadores de teste do Laboratório Bridge da UFSC (Universidade Federal de Santa Catarina).

3 OBJETIVOS

Esta seção tratará dos objetivos e metas a serem alcançados através da pesquisa e execução do trabalho de conclusão de curso.

3.1 Objetivo geral

Esta pesquisa tem como objetivo propor, aplicar e validar as respostas obtidas no questionário avaliativo a respeito da nova ferramenta, desenvolvida para resolver os problemas relacionados à manutenção dos casos de teste automatizados vinculados aos requisitos do *software*.

3.2 Objetivos específicos

Além disso, esta pesquisa tem como objetivo estudar todos os conceitos relacionados a este assunto, englobando metodologia ágil, análise de requisitos e teste de *software*. Como objetivos específicos para este trabalho pode-se citar:

- Foram propostas operações mínimas que satisfaçam a resolução do problema identificado no laboratório Bridge.

- A partir das operações mínimas propostas, foi desenvolvida uma ferramenta que busque auxiliar na rastreabilidade entre requisitos documentados e seus casos de testes automatizados .

- Com a ferramenta desenvolvida, os autores aplicaram a solução proposta aos testadores do Laboratório Bridge.

- Após a aplicação da solução, um questionário foi realizado com os testadores a fim de extrair conclusões sobre a ferramenta, tais como: usabilidade, utilidade e êxito ou fracasso no cumprimento de sua principal função.

4 JUSTIFICATIVA E MOTIVAÇÃO

Durante o aprendizado no Laboratório Bridge, foi identificado uma lacuna que faltava ser preenchida entre duas áreas do desenvolvimento de *software*: a rastreabilidade entre a documentação do *software*, suas especificações, produzida pela área da análise, e os casos de teste, produzidos pelo setor de qualidade. Este problema tem um impacto maior nos casos de testes produzidos pelos testes automatizados. Muitas vezes, com uma troca de regra de negócio gerada por um requisito, muito casos de teste automatizados acabavam “quebrando” por não se aplicarem mais ou porque sua especificação mudou, gerando muito retrabalho. Todos os casos de testes deveriam ser revistos por não se saber em quais casos aquela alteração impactaria.

Buscamos uma solução para facilitar essa rastreabilidade entre as alterações feitas na documentação e seus casos de testes relacionados, para uma manutenção menor, mais rápida, simples e eficaz.

5 PROCEDIMENTOS METODOLÓGICOS

A metodologia de pesquisa aplicada neste trabalho engloba todo o processo de realização do trabalho, desde a concepção do tema e da ideia principal até a conclusão do mesmo. Os métodos utilizados para desenvolvimento da pesquisa envolvem a leitura de artigos e livros relacionados ao tema, bem como a compreensão dos mesmos, a relação destes com o estado da arte do tema escolhido e reuniões com profissionais da área de testes e qualidade de *software* do Laboratório Bridge.

A partir do resumo do estado da arte e da vivência profissional dos autores deste trabalho, foi possível perceber quais problemas que existem na área, bem como melhorias que precisam e podem ser feitas. Além disso, faz parte do processo de criação do trabalho tentar achar novas propostas de sistemas que podem utilizar as mesmas soluções já encontradas nos artigos analisados anteriormente. Com o problema definido, não foram encontradas soluções que o resolvam de maneira direta, fazendo com que uma nova solução fosse proposta e analisada.

O método de pesquisa se estende além da leitura e compreensão de artigos, e vai até a análise de códigos já existentes que tratam do mesmo problema ou de problemas semelhantes ao exposto no trabalho. Após definir um problema específico, foram programadas reuniões com o orientador para definir uma implementação possível para o tema e o problema em questão. A

implementação utiliza o conhecimento adquirido pelas disciplinas que envolvem programação e engenharia de *software* na graduação.

Afim de verificar a eficácia da ferramenta proposta, uma pesquisa em forma de questionário foi elaborada, aplicada e analisada junto à solução prática.

5.1 Estrutura do documento

O projeto de pesquisa foi realizado em parceria com o Laboratório Bridge da Universidade Federal de Santa Catarina.

O escopo do trabalho foi definido de acordo com a especificação e regras definidas pela Biblioteca Universitária. O trabalho é composto por uma introdução, que explica detalhadamente a motivação por trás do tema escolhido e os objetivos gerais e específicos da pesquisa realizada. Contêm também capítulos de conceitos básicos relacionados à área da aplicação e que, portanto, são explicados a qualquer usuário (leitor) do trabalho e do processo que foi desenvolvido. O trabalho contém uma visão ampla do estado da arte na área de engenharia de *software*.

O trabalho é uma pesquisa e, portanto, busca contribuir para a área científica. O capítulo de desenvolvimento explica isso através da elaboração de uma solução visando a rastreabilidade entre testes automatizados e a documentação de requisitos do *software*.

Após o desenvolvimento destes capítulos, é apresentada uma conclusão

do que foi aprendido durante a pesquisa e os anos de graduação dos alunos e uma proposta para trabalhos futuros.

6 FUNDAMENTAÇÃO TEÓRICA

6.1 Introdução à engenharia de *software*

Entre as décadas de 60 e 70 houve uma grandiosa expansão da capacidade computacional do *hardware*, fazendo com que surgisse um aumento na demanda por *software* mais complexo capaz de utilizar tais capacidades a sua disposição. Desta maneira, o mercado passou a tratar o *software* como um produto comercializável capaz de ser desenvolvido para solucionar diferentes tipos de problemas [16]. Contudo, em tal período, as técnicas utilizadas para o desenvolvimento de tais soluções não eram as mais adequadas para suprir as demandas, exigências de qualidade e prazos de entrega.

> *"A maior causa da crise do software é que as máquinas tornaram-se várias ordens de magnitude mais potentes! Em termos diretos, enquanto não havia máquinas, programar não era um problema; quando tivemos computadores fracos, isso se tornou um problema pequeno e agora que temos computadores gigantescos, programar tornou-se um problema gigantesco." (Edsger Dijkstra na sua apresentação "The Humble Programmer").*

Diversos problemas para o processo de desenvolvimento de sistemas foram acarretados devida a demanda de *software* mais avançado, gerando a "crise do *software*" [16].

As metodologias de construção de *software* estavam causando entregas completamente fora dos prazos estabelecidos, estouro de orçamentos e grau de qualidade muito inferior ao esperado. De acordo com o relatório de Naur [7], cerca de 50% a 80% dos projetos nunca foram concluídos ou estavam tão longe de seus objetivos que foram considerados fracassados. Dos sistemas que foram finalizados, 90% haviam terminado 150% a 400% acima do orçamento e dos prazos predeterminados [4].

Estes problemas eram e, ainda hoje são, sintomas provenientes do pouco entendimento dos requisitos por parte dos desenvolvedores, somados às técnicas e medidas pobres aplicadas sobre o processo e o produto, além dos poucos critérios de qualidade estabelecidos até então [5].

Tal crise foi responsável por uma revisão dos conceitos utilizados e técnicas vigentes, resultando na criação de ferramentas, metodologias e mudanças nos pensamentos. Assim, criou-se o conceito de Engenharia de *Software*.

A Engenharia de *Software* é composta por métodos, ferramentas e procedimentos. Ela possibilita ao gerente do projeto controlar todo o processo de desenvolvimento, e ainda oferece ao programador um apoio para a implementação de um produto com qualidade [5]. Segundo Pressman [5], estes

paradigmas contêm três fases genéricas que são encontradas em todo desenvolvimento de *software*: definição, desenvolvimento e manutenção.

6.2 Modelos de desenvolvimento de *software*

Para a construção de um *software*, diferentes modelos de desenvolvimento foram arquitetados com o intuito de padronizar e prover auxílio às suas produções. Há uma generalização em tais modelos que podem ser observadas nas seguintes camadas: comunicação, planejamento, modelagem, construção e implantação. Contudo, para cada camada pode ser dado um enfoque diferente, ou ainda ela ser invocada de maneira diferente, dependendo do modelo em questão [5].

Diferentes itens intermediários são produzidos até que se chegue no produto final, como por exemplo: dados de carga, manuais, análises, relatórios, testes, componentes, formulários, planos de trabalho, especificações, cenários de uso, glossários, diagramas UML, entre outros. Genericamente, tais elementos são denominados artefatos. Estes são criados ou atualizados em diferentes fases da produção de um *software* juntamente com um conjunto de ferramentas, que deve ser usado para armazenar dados futuramente empregados em análises ou servidos como uma parte da documentação [5].

Os documentos exercem duas funções específicas: atuam como registros históricos das decisões do projeto e também como manuais explicativos do

código, funcionamento do sistema e regras de negócio. A documentação possui forte influência na comunicação da equipe desenvolvedora e suas interações com os *stakeholders* [17]. Porém, na prática, a comunicação através de documentos dificilmente substitui a comunicação verbal e as interações entre pessoas. Dois níveis de interações têm sua criação incentivada pela utilização de documentos: oficial, que é feito através do conjunto de artefatos produzidos para o projeto; e outro informal, onde estão as relações interpessoais, englobando os bate-papos, conversas ao telefone, troca de emails e pequenas discussões [18].

6.3 Desenvolvimento ágil

Com a evolução e troca de experiências dos processos de Engenharia de *Software*, modos de trabalho foram adotados se opondo aos principais conceitos das metodologias tradicionais. Tais modos passaram a ser chamados de leves e formaram novas metodologias que não utilizavam as formalidades que caracterizam os processos tradicionais, evitando a burocracia imposta na utilização excessiva de documentos.

Desta maneira, foram identificados 12 princípios determinantes para a obtenção de bons resultados publicados no Manifesto Ágil [8], representado por quatro premissas:

- **Indivíduos e interações** são mais importantes que processos e ferramentas

- ***Software* em funcionamento** é mais importante que documentação abrangente
- **Colaboração com o cliente** é mais importante que negociação de contratos
- **Responder a mudanças** é mais importante que seguir um plano

A implantação de uma metodologia ou partes de seus conceitos é algo muito trabalhoso e visto como uma grande demanda de esforço ao desenvolvimento.

Mudar padrões já estabelecidos e adotar uma metodologia, geralmente, não é bem vista dentro da equipe de desenvolvimento. Aos olhos da equipe, pode ser notado uma piora no desempenho logo de início.

As metodologias ágeis buscam a geração de documentos indispensáveis ao projeto, sem que haja excessos de criação e/ou entendimentos destes. Propondo a obtenção de resultados práticos em um período menor do que a indústria de *software* estava acostumada, as metodologias ágeis tiram o foco do processo e o colocam no produto. Tal construção promove uma base de conhecimento que será usada no futuro para melhoria dos processos e para a própria reorganização da metodologia. Proporcionando, assim, um vislumbramento mais nítido dos pontos e processos em déficit de progresso [5].

O estudo de metodologias, principalmente as ágeis, leva a acreditar que a adoção de um método, ou parte dele, dentro do processo de desenvolvimento só vem a contribuir para a otimização dos esforços dos recursos, tanto humanos como financeiros, na organização.

6.4 Análise de requisitos

O levantamento de requisitos é o processo de descobrir quais são as funções que o sistema deve realizar e quais são as restrições que existem sobre essas funções. Operações que venham a constituir a funcionalidade do sistema são chamadas de requisitos funcionais. As restrições sobre essas operações são conhecidas como requisitos não funcionais [19].

O início para toda a atividade de desenvolvimento de *software* é o levantamento de requisitos, sendo esta atividade repetida em todas as demais etapas da engenharia de requisitos.

De acordo com De Pádua Paula Filho [27], a boa engenharia de requisitos reduz a instabilidade destes, ajudando a obter os requisitos corretos em um estágio anterior ao desenvolvimento. Entretanto, alterações dos requisitos são às vezes inevitáveis. A engenharia de requisitos é sujeita a limitações humanas, e mesmo que o levantamento seja perfeito, podem ocorrer alterações de requisitos por causas externas aos projetos. Por exemplo, a legislação pode mudar no meio do projeto, requerendo alterações nos relatórios que o produto deve emitir.

Sommerville [10] propõe um processo genérico de levantamento e análise que contém as seguintes atividades:

- Compreensão do domínio: Os analistas devem desenvolver sua compreensão do domínio da aplicação;

- Coleta de requisitos: É o processo de interagir com os *stakeholders* do sistema para descobrir seus requisitos. A compreensão do domínio se desenvolve mais durante essa atividade;
- Classificação: Essa atividade considera o conjunto não estruturado dos requisitos e os organiza em grupos coerentes;
- Resolução de conflitos: Quando múltiplos *stakeholders* estão envolvidos, os requisitos podem apresentar conflitos. Essa atividade tem por objetivo solucionar esses conflitos;
- Definição das prioridades: Em qualquer conjunto de requisitos, alguns serão mais importantes do que outros. Esse estágio envolve interação com os *stakeholders* para a definição dos requisitos mais importantes;
- Verificação de requisitos: Os requisitos são verificados para descobrir se estão completos e consistentes e se estão em concordância com o que os *stakeholders* desejam do sistema.

A análise de requisitos é definida pela identificação das funcionalidades desejadas para o *software*. É necessário um conhecimento detalhado sobre as regras de negócio e o domínio que o sistema se propõe a tratar. As necessidades de interface, interação e demais fatores que determinam a maneira como outros sistemas e usuários utilizarão o *software* devem ser identificados, documentados e acordados com o cliente [20].

Com as entregas frequentes definidas pelas metodologias ágeis, conforme os requisitos surgem, há a atualização do *software*. Cada versão entregue deve

ter o menor tamanho possível, contendo os requisitos de maior valor para o negócio [9].

O amadurecimento das práticas ágeis proporcionou o entendimento de que os requisitos são mutáveis e suas alterações devem ser bem-vindas. Dentre os fatores responsáveis por alterações nos requisitos estão a dinâmica das organizações, as alterações nas leis,as mudanças pedidas pelos *stakeholders*, que geralmente têm dificuldades em definir o escopo do futuro *software* ou a necessidade de alterações nas regras de negócio de versões já em produção [13].

Com o intuito de minimizar os impactos causados pelas mudanças nos requisitos e estar preparado para atendê-las, o *feedback* constante do cliente é necessário.

6.5 Teste de *software*

Teste de *Software* é um conjunto de atividades antecipadamente planejadas e executadas sistematicamente para garantir a qualidade de um *software* [16]. Definindo um roteiro de teste, passos para execução, com estratégias e técnicas diferenciadas, conseguimos garantir que o produto atingiu vários aspectos especificados como: segurança, desempenho e confiabilidade; e executou corretamente para o propósito que foi projetado. Como nem sempre é possível deixar o *software* livre de defeitos [11], ele tem como objetivo expor o

maior número de defeitos que o mesmo possui, para que possam ser tratadas em versões futuras.

6.5.1 Conceitos básicos sobre teste de *software*

Alguns conceitos que devemos saber sobre teste de *software* são [16]:

- **Falha:** Produção de uma saída incorreta com relação à especificação. A falha é a percepção do erro.
- **Erro:** Diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro.
- **Defeito:** Passo, processo ou definição de dados incorretos, como por exemplo, uma instrução ou comando incorreto.
- **Engano:** Ação humana que produz um resultado incorreto, como uma ação incorreta tomada pelo programador.
- **Caso de Teste:** Descreve uma condição particular a ser testada e é composto por valores de entrada, restrições para a sua execução e um resultado ou comportamento esperado.

6.5.2 Técnicas de teste de *software*

Para a execução de Testes de *Software*, existem várias estratégias, técnicas e tipos de teste que dizem o que testar, como testar e quando testar. São eles [21]:

- Técnicas de teste, ou “como Testar”:
    - Técnica de Caixa-Branca
    - Técnica de Caixa-Preta

- Tipos de teste, ou “o que testar”:
    - Teste de Regressão
    - Teste de Usabilidade
    - Teste de Desempenho (Carga, estresse, resistência)
    - Teste de Configuração
    - Teste de Confiabilidade
    - Teste de Recuperação
    - Teste de Instalação
    - Teste de Segurança

- Níveis de teste, ou “quando testar”:
    - Teste de Unidade

    - Teste de Integração
    - Teste de Sistema
    - Teste de Aceitação

- Abordagem de teste:
    - Teste Exploratório
    - Teste Automatizado

Destes tipos, técnicas, níveis e abordagens, é interessante para este trabalho o Teste Automatizado e o Teste de Regressão, que serão abordados nas seções seguintes.

6.5.2.1 Teste de regressão

Teste de Regressão é uma técnica de Teste de *Software* que consiste em aplicar novamente em uma versão mais nova do sistema, todos os testes feitos anteriormente. Quando uma nova funcionalidade, módulo ou correção é adicionado ao sistema, o teste de regressão garante que não surgiram novas falhas em componentes já analisados previamente. Para a execução do teste de regressão, geralmente são utilizadas ferramentas para automatizar esses testes.

6.5.2.2 Teste automatizado

Para Testes Automatizados, são utilizadas ferramentas de *software* para a elaboração, execução e controle dos testes. A partir de testes de *software* já consolidados, podemos automatizá-los, deixando a cargo da ferramenta, e não do usuário, a execução e verificação dos mesmos, facilitando a execução repetitiva do mesmo teste várias vezes. Como a execução dos testes de *software* pode ser um trabalho maçante, a escrita de testes automatizados permite que uma vasta gama de casos de testes podem ser validados rapidamente.

A implantação de testes automatizados pode ser custosa no início, mas a longo prazo, para *software* de grande porte ou que estão em produção durante muito tempo, economiza muito tempo de teste, principalmente em testes de regressão.

Para a automação de testes, duas abordagens podem ser utilizadas: testes baseados na interface gráfica do usuário e testes baseados em código. Para testes baseados na interface gráfica do usuário, uma ferramenta de *software* simula as entradas no *software* que se quer testar como um usuário faria. Essas entradas são definidas pelo programador do teste, com base nos artefatos criados previamente, como os casos de teste, assim observando as ações esperadas. Em nível de código, o programador do teste constrói o teste direto no código, possibilitando os testes de unidade, que foca em seções do código do *software* testado, observando se estão executando da forma esperada.

7 PROBLEMA

Durante o tempo de estágio no Laboratório Bridge, identificamos um problema recorrente que dificultava o trabalho dos testadores: a revalidação de seus casos de teste quando ocorria uma alteração na documentação do *software*. Tal problema se amplifica quando se trata dos testes automatizados: por serem muitos testes, por não sabermos especificamente qual teste engloba qual regra, por não sabermos qual regra mudou e se essa mudança foi significativa a ponto de que uma refatoração do código do teste automatizado seja necessária.

Para o automatizador de teste, cujo testes automatizados são escritos baseados nas regras do *software* descritos na documentação, no caso do Laboratório Bridge, a incerteza nas mudanças das regras dificulta o processo de refatoração dos testes automatizados. Uma refatoração preventiva é inviabilizada, visto que o automatizador só saberá da mudança na regra quando o teste automatizado referente aquela regra for executado e retornar “sem sucesso”. No cenário de testes automatizados do Laboratório, um teste automatizado retornará “sucesso” na sua execução quando não houver problemas nas verificações feitas pelo mesmo, e retornará “sem sucesso” quando encontrar um problema nas verificações implementadas em seu código.

Um problema maior é quando o teste automatizado de uma regra que sofreu alteração retornar “sucesso”. Neste caso, o teste automatizado se torna insuficiente, não testando o que ele deveria abranger da regra, deixando o

processo de qualidade do *software* defasado por não garantir que aquela regra foi corretamente implementada no *software*. O automatizador de teste não saberá que tal teste automatizado está defasado por sempre retornar “sucesso” na sua execução.

Esse processo de refatoração dos testes sob demanda é considerado ineficiente levando em conta o porte dos *software* desenvolvidos pelo Laboratório. São necessários muitos testes automatizados para garantir a qualidade do produto. Como descrito anteriormente, os testes são baseados nas regras do *software* e tais regras estão sujeitas a mudanças constantes, gerando uma grande demanda na refatoração dos testes automatizados para a equipe responsável.

É um processo custoso para o automatizador descobrir manualmente quais regras foram alteradas e quais testes automatizados devem ser revalidados pois, a mudança de regras só será visível utilizando a ferramenta *git*[33], onde a documentação do *software* está hospedada.

No Laboratório Bridge, o código dos testes automatizados está organizado na forma de pacotes. Cada pacote é referente a um módulo do *software* que ele testa, dentro dele encontram-se as classes de testes automatizados. Para descobrir qual teste automatizado abrange determinada regra, o automatizador deverá navegar na estrutura de pacotes para achar qual teste automatizado refere-se à regra desejada.

Percebe-se que é um processo demorado, complexo, exaustivo, repetitivo e ineficiente, por isso não é feito no Laboratório. Pelos motivos apresentados, desenvolvemos uma ferramenta que facilita o processo de rastreabilidade.

8 SOLUÇÃO

Pensando sobre como poderíamos solucionar esse problema, identificamos três operações que são necessárias para que a solução satisfaça nossas necessidades:

1. Obter uma ligação entre o texto da documentação e o código do teste automatizado, que chamamos de *trace* ou “rastro”.
2. Identificar se ocorreu uma mudança no texto da documentação, que chamamos de *verify* ou “verificação”.
3. Validar se a mudança identificada do texto impactou no teste e se ele deve ser refatorado, que chamamos de *“check”* ou “validar”.

A operação número um, *trace*, foi automatizada através da implementação de *“tags”*, códigos de identificação únicos, que representam um texto da documentação. Essa mesma *tag* deve ser incluida no código do teste automatizado, na forma de um comentário, para que o *link* seja feito.

A operação número dois, *verify,* foi automatizada comparando o texto da regra em um estado inicial, com o texto da mesma regra oriundo de uma mudança. Assim, unindo a operação um e dois, conseguimos identificar e mostrar ao automatizador de testes qual regra mudou e qual teste automatizado deve ser reavaliado.

A operação número três, *check*, deve ser uma operação humana. Com o resultado das operações um e dois, o universo de testes automatizados e regras

de documentação são reduzidos a somente aqueles que devem ser reavaliados. Assim, o automatizador de testes pode reavaliar os testes automatizados um por um, refatorando o que for necessário e validando quais as regras sofreram alteração mas não geram impacto no código do respectivo teste.

8.1 Desenvolvimento e execução da ferramenta

O desenvolvimento foi focado na solução do problema encontrado no Laboratório Bridge, de forma que a ferramenta interfira o mínimo possível no trabalho de outras pessoas que não são automatizadores de testes e no dos mesmo. A prévia utilização de outras ferramentas pelas partes interessadas, análise de *software* e qualidade de *software*, possibilitou uma abordagem para a codificação, descrita abaixo.

A execução da ferramenta foi dividida em três etapas. A preparação para que a rastreabilidade seja feita, o estabelecimento do estado dos requisitos, e a comparação dos estados dos requisitos. Os estados dos requisitos são definidos como estado alterado ou não alterado.

Com o conhecimento adquirido durante as disciplinas de programação cursadas na graduação, foi possível desenvolver a solução codificada em linguagem *Java* [22]. Analisando como poderíamos usar os artifícios da linguagem para codificar a solução, optamos por uma manipulação de arquivos.

A preparação começa com a definição das *tags* e sua inserção na documentação e no código das classes de testes automatizados. O usuário da ferramenta também deve informar em qual diretório do computador estão armazenadas as classes de testes automatizados e os arquivos da documentação do *software*.

Os artefatos produzidos pela análise de requisitos do Laboratório Bridge estão todos descritos em texto com *markdown* [23], possibilitando que sejam inseridos comentários no texto com as *tags* para gerar o identificador único da regra, no começo e no fim do trecho de texto que se deseja referenciar. Como são comentários e estão escritos em *markdown*, sua presença não impactará na leitura e utilização das outras pessoas que consomem esses artefatos, pois o comentário não aparecerá para o leitor. Sugerimos que para uma melhor utilização, cada trecho delimitado e definido pelas *tag* devem corresponder a um requisito do *software* ou parte do requisito.

As classes de código de teste automatizados do Laboratório Bridge são código *Java* com a utilização do *framework Selenium* [34]. Estas classes também podem ser tratadas pelo *Java* como um simples arquivo de texto, assim também possibilitando a inserção da *tag* em forma de comentário. Os comentários foram escolhidos como solução por não gerarem impacto na execução do código e na leitura dos requisitos do *software* documentado.

Com a inserção das *tags* nos arquivos da documentação e nos arquivos das classes de testes automatizados, a primeira etapa da execução da

ferramenta, a preparação para o rastreio, se encerra. A segunda etapa, de estabelecimento do estado dos requisitos, começa a ser descrita abaixo.

Com esses comentários, nos arquivos da documentação e arquivos das classes dos testes automatizados, e a manipulação de arquivos do *Java*, a solução codificada consegue ler todos os arquivos da documentação do *software*, a partir de um diretório especificado e procurar pelos trechos de texto marcados com as *tags.* Identificando as *tags,* os trechos de texto são armazenados em um *array* de *Strings* para manipulação futura. Os trechos de texto de requisitos de cada *tag* marcada na documentação armazenados no *array*, então são transformados em um *hash* codificado em SHA256 [26] como artifício computacional para diminuir o tamanho do texto armazenado e para facilitar a comparação futura. Apenas o texto da documentação é transformado em *hash*.

Do mesmo modo que são tratados os arquivos da documentação, a solução consegue ler todos os arquivos das classes de código dos testes automatizados, linha por linha, procurando e identificando as *tags* nos comentários, armazenando o nome da classe do teste automatizado, referente a cada *tag,* em um *array* de *Strings.*

Com isso, a solução une, através da *tag* que está nos arquivos de testes automatizados e na documentação, em um único arquivo de texto*,* a *tag* identificadora, o *hash* do trecho de texto correspondente ao requisito, e o nome da classe do código do teste automatizado em uma única linha. Cada linha do arquivo de texto contém uma *tag*, um *hash* e o nome de uma classe. Este arquivo

contém o estado atual da rastreabilidade das regras da documentação com seus respectivos testes automatizados.

Assim é encerrada a segunda etapa de execução da ferramenta, que mapeia cada regra definida pelas *tags* na etapa anterior, com a classes de testes automatizados correspondentes, também identificadas pelas *tags* anteriormente.

A terceira etapa de execução inicia-se quando houver a necessidade de identificar se houve uma mudança em um requisito rastreado. No Laboratório Bridge, o processo de atualização da documentação do *software* nunca deixou muito claro, explícito, o que foi alterado de uma versão para outra. Para o automatizador de teste, que tem seus testes automatizados escritos baseados na documentação, essa dúvida pode ser sanada somente acessando a ferramenta *git* [33], e comparando as versões através das utilidades da ferramenta, conseguindo ver o que foi alterado. Isto é um processo custoso para o automatizador pois, como há muitos analistas produzindo e alterando regras, muitas regras podem ter sido alteradas. Como explicado no capítulo anterior, esse processo de descoberta das mudanças geradas não é feito.

Ao concluir as etapas um e dois, temos um arquivo de texto contendo a rastreabilidade da documentação com o teste automatizado, explicado anteriormente. Com o *hash* do texto da documentação feito na etapa dois, temos o estado atual da regra rastreada.

No Laboratório Bridge, o uso da ferramenta *git* facilita o versionamento e trabalho dos analistas por possibilitar que todos alterem os mesmos arquivos simultaneamente, assim quando há uma alteração feita por algum analista, todos

terão acesso a essas mudanças em seus arquivos. Do mesmo modo, os automatizadores de teste, que utilizam a documentação produzida pelos analistas, recebem as mudanças.

Para identificar se houve uma mudança, primeiramente o automatizador de testes deve receber as mudanças vindas do *git.* A solução irá gerar novamente um arquivo de estado das regras, como descrito anteriormente, porém usará os arquivos modificados. Sabendo que as *tags* não foram alteradas de uma versão para outra, como explicado posteriormente na seção de problemas encontrados, um arquivo novo de estado das regras alteradas será gerado, como vimos.

Estando em posse do arquivo de estado das regras e do arquivo de estado das regras alteradas, a solução irá comparar *tag* por *tag* se o *hash* do trecho de texto, referente a uma regra e identificado por uma *tag,* está diferente. Se o *hash* do estado da regra for diferente do estado da regra alterada, isso implica que o trecho de texto rastreado na documentação pelas *tags* sofreu alteração.

A solução então irá listar todas as *tags* cujo *hash* referente foi diferente nos arquivos comparados, exibindo para o automatizador de testes o nome da classe do teste automatizado referente a regra da documentação que ele abrange, referenciado pela *tag.*

O arquivo de estado das regras alteradas passa a ser tratado como o estado atual das regras, e é armazenado para futuramente ser comparado novamente com um novo arquivo de estado das regras alteradas, finalizando o processo de execução da solução.

Sendo assim, a solução irá poupar o automatizador de teste de todo o trabalho necessário para descobrir previamente quais testes automatizados devem ser revalidados por causa de uma modificação em sua regra, facilitando, simplificando e encurtando o processo que vimos anteriormente.

O grande impacto no processo do automatizador de teste será implantar as *tags* no primeiro momento e posteriormente mantê-las quando forem criadas novas regras na documentação que se deseja testar automatizadamente. Para o analista de *software*, em seu processo de trabalho ele deverá zelar pela integridade das *tags* inseridas pelo automatizador, assim entrando em comum acordo para que os dois elaborem um bom trabalho sem interferir no trabalho do seu colega.

Alguns exemplos de uso podem ser vistos abaixo:

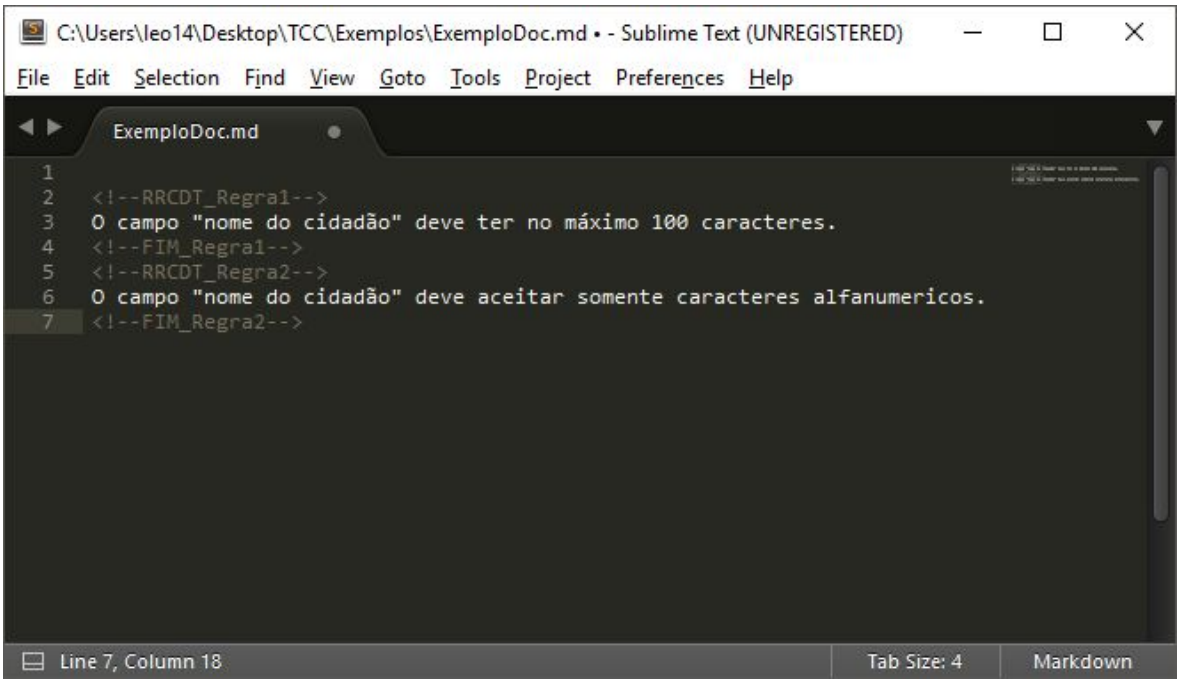

Figura 2 - Exemplo Tag Documentação

Como podemos observar na figura acima, um exemplo da utilização das *tags* em um exemplo de texto de documentação de software. A *tag* divide os requisitos descritos no texto da documentação.

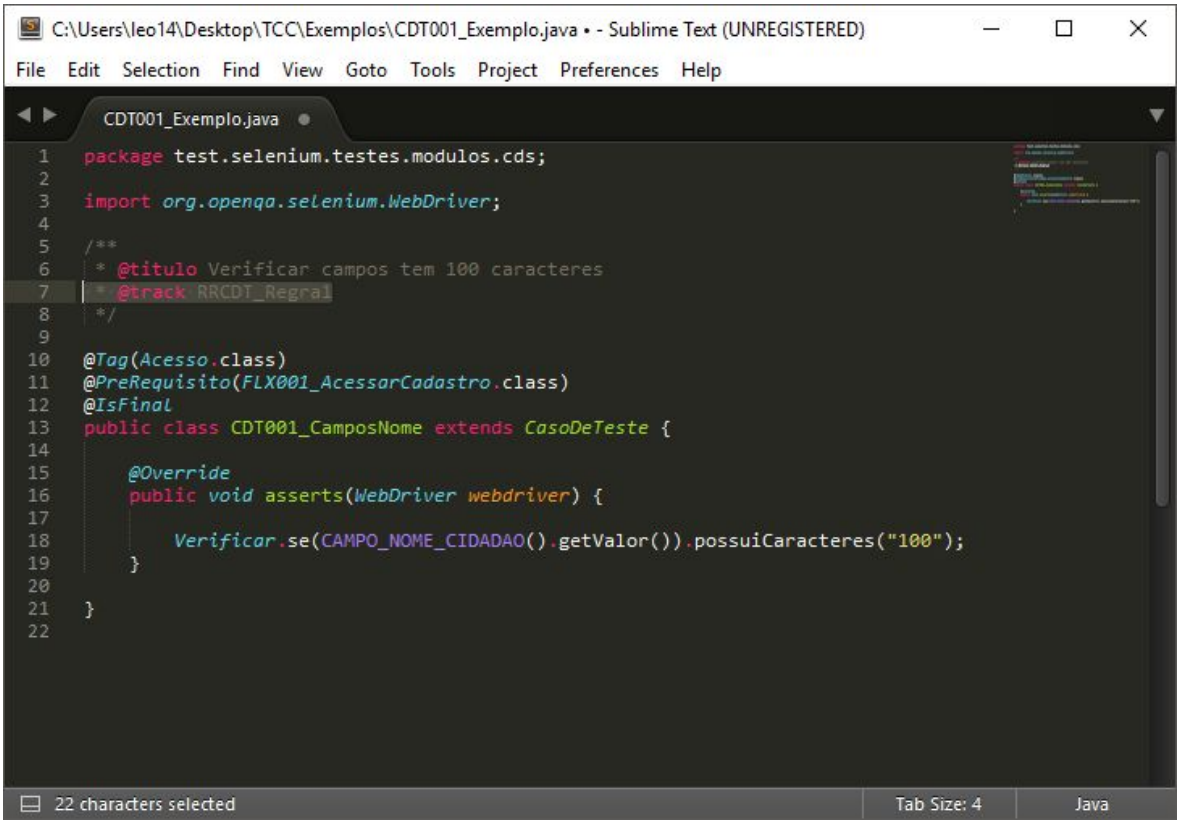

Figura 3 - Exemplo Tag Caso de Teste

Na figura acima, temos um exemplo de classe de teste automatizado, escrito em *JAVA*. Podemos notar o uso da *tag* na linha 7, onde ela é inserida na forma de um comentário, precedida por "@*track*" para melhor identificação.

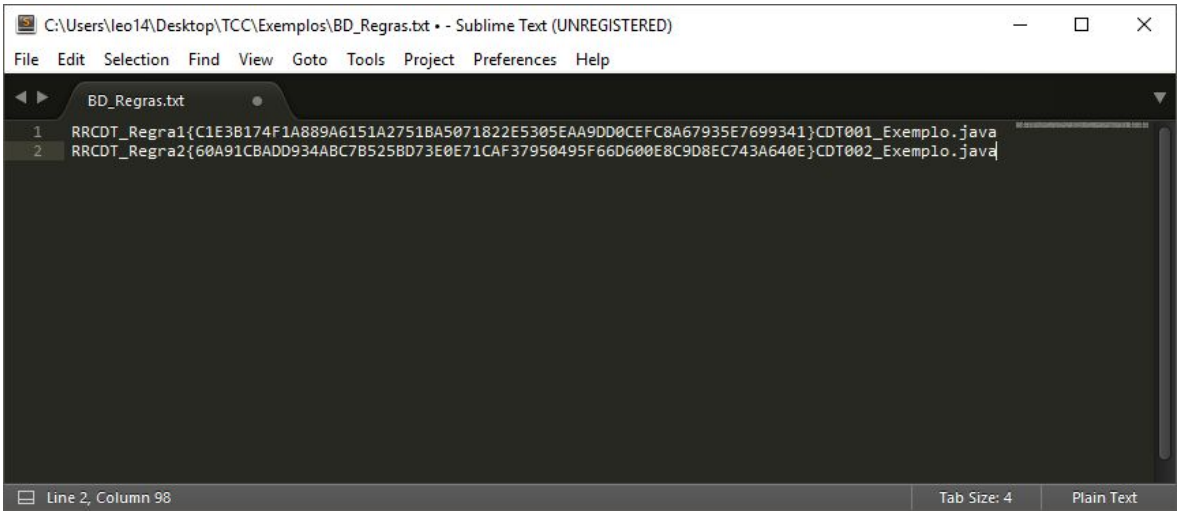

Figura 4 - Exemplo de arquivo de estado das regras

Na figura acima podemos ver um exemplo do arquivo criado para guardas as regras de rastreio. Podemos ver cada linha, representando um conjunto de *tag, hash* e o nome do arquivo da classe *JAVA* do teste automatizado.

Nas imagens acima podemos ver exemplos de uso das *tags* em regras que descrevem o sistema e o uso nas classes *Java* de testes automatizados, bem como ficará o arquivo de estado das regras.

8.2 Problemas encontrados

Durante o desenvolvimento e execução do *software* alguns problemas foram encontrados.

Como delimitado no escopo do trabalho, a adição de novos trechos de texto na documentação de *software* que não estejam relacionados a nenhuma *tag*

já descrita não serão encontrados e exibidos para o automatizador de teste, assim como a exclusão de trechos de texto juntamente com a *tag* que identificá, também não será mais rastreado nem apresentado nos resultados de mudanças. Uma possível solução para este problema é o uso de um *hash* do conjunto do arquivo, como é explicado na seção de trabalhos futuros.

O mal uso da ferramenta também impacta na execução da mesma, como não colocar ou retirar a *tag* que delimita o fim ou o começo do trecho de texto que deseja ser identificado. Deve ser acordado entre as partes que utilizam a ferramenta e modificam os mesmos arquivos para sempre manter a integridade das *tag*, procurando apenas alterar somente o trecho de texto delimitado pelas *tags*.

Como a *tag* tem o intuito de funcionar como um identificador único, repetir a mesma *tag* para mais de um trecho de texto pode causar confusão no entendimento dos resultados.

Como a solução serve para identificar trechos de texto, é possível que a *tag* seja inserida no meio de uma frase, contudo isso impactará no trabalho dos profissionais da análise de *software*, onde o texto que eles lerão estará “poluído” com *tag* no meio das frases. Relembrando que a *tag*, por ser um comentário, não aparecerá para os usuários que consomem a documentação do *software*, mas para os profissionais que escrevem a documentação e tem contato direto com o texto em *markdown*, poderão sofrer com esse possível problema. Recomendamos inserir as *tags* uma linha acima e uma linha abaixo do texto que se deseja identificar.

Outro problema identificado é que a solução faz um mapeamento “um para um”. Uma *tag* está associada a um trecho de texto e a uma classe de teste automatizado. Vimos no processo de trabalho no Laboratório Bridge que uma classe de teste automatizado pode testar mais de uma regra, ou seja, mais de uma *tag* poderia estar associada a uma classe de teste automatizado. Este problema surgiu por uma falta de planejamento no desenvolvimento da solução, onde não foi previsto abrangência de mais de uma regra por um teste automatizado.

Percebemos que as classes em *Java* dos testes automatizados, por mais que não possam ter o mesmo nome dentro de um *package*, em outros *package* pode haver uma classe com o mesmo nome, assim também irá tornar o entendimento da saída do resultado da execução da solução dubio. Uma possível saída é ao invés de usar somente o nome da classe, usar também o seu *path* para distinguir classes com o mesmo nome.

Ao executar a solução, percebemos que a operação número três, *check*, descrita anteriormente, ficou subentendida no fluxo da ferramenta, assumindo que todos os testes automatizados apresentados para reavaliação serão validados pelo automatizador de teste antes da próxima execução.

8.3 Outras ferramentas disponíveis

Pesquisamos outras possíveis soluções para o problema encontrado antes de iniciar o desenvolvimento, mas nenhuma apresentava o comportamento desejado: automatizar a descoberta das mudanças e apresentar para o automatizador somente o necessário para o seu trabalho.

Analisando a ferramenta Matriz de Rastreabilidade [29], percebemos que é uma ferramenta que supre a necessidade de rastreabilidade, porém não provê o suporte automatizado necessário para uma descoberta ágil do que mudou na documentação do *software* e o que deve ser mudado nos testes automatizados. Servindo mais como um plano de cobertura de testes, para que o testador consiga rastrear os pontos que foram cobertos por testes.

Analisando a ferramenta Prometeu [30], percebemos que é uma ferramenta completa e complexa, implementando 8 tipos de documentos de teste e artefatos produzidos para a atividade de teste de *software*. Desta forma, a adoção de tal solução necessita de uma mudança no fluxo, na metodologia e na interação do ambiente de trabalho, sendo custoso. Baseada na norma IEEE-Std-829 [31], ela implementa uma série de documentos que dão suporte ao teste de *software.* Por ser muito complexa, os documentos produzidos pelo teste de *software* ficam todos centrados nela, impactando em uma mudança muito radical do processo de teste e análise de todo o laboratório. Procuramos uma ferramenta que se encaixe na realidade do problema. Toda a documentação estaria hospedada nela, mudando o fluxo de trabalho dos analistas. Com a solução proposta, o trabalho dos analistas não sofrerá um grande impacto pois tudo será feito pelo automatizador de testes.

A ferramenta Cucumber [31] usa um modelo diferente de teste abordado no Bridge, local onde evidenciamos o problema. Ela é baseada em BDD, que utiliza uma linguagem com sintaxe de descrição de comportamento em texto plano. Facilita para que todas as pessoas envolvidas no desenvolvimento de *software* entendam qual o comportamento do sistema, qual o comportamento do teste automatizado e o que o desenvolvedor deve implementar, pois a partir do texto documentado, o teste é escrito e, sobre o mesmo é implementada a funcionalidade. A rastreabilidade das funcionalidades com o teste será suprida, porém não temos o suporte automatizado das mudanças que desejamos.

9 Experimento

Para avaliar a proposta de solução apresentada, utilizamos o Laboratório Bridge como experimento para aplicar a solução e colher dados para elaborar uma conclusão sobre a efetividade da ferramenta.

9.1 O laboratório Bridge

O Bridge é um laboratório integrado ao Centro Tecnológico (CTC) da Universidade Federal de Santa Catarina (UFSC), que atua na pesquisa e

desenvolvimento de soluções tecnológicas conectando governo e cidadão por meio da inovação [28].

Atualmente no Bridge, são desenvolvidos três grandes projetos nacionais de informatização que estão ajudando na qualificação da gestão pública, o e-SUS AB (Atenção Básica) e o SISMOB (Sistema de Monitoramento de Obras) ambos do Ministério da Saúde, e o RNI (Registro Nacional de Implantes) da ANVISA [28].

No Bridge, o desenvolvimento de *software* é dividido em equipes. Há uma equipe de teste automatizado que provê suporte aos seus projetos. Juntamente com esta equipe, dentro das equipes ágeis, os testadores ágeis também ajudam a criar testes automatizados para as tarefas desenvolvidas dentro de cada equipe. O trabalho de manutenção destes testes automatizados criados pelos testadores das equipes ágeis e a própria equipe de teste automatizado é responsabilidade desta.

9.1.1 O processo

No Bridge, testes automatizados são utilizados principalmente para execução de Testes de Regressão, como explicado anteriormente, garantindo que uma nova implementação no *software* não impactou em outras partes que estavam funcionando conforme o esperado antes desta nova implementação.

Analisando a equipe de teste automatizado, atualmente a equipe não possui processo definido para a manutenção prévia dos testes automatizados.

Para identificar que um teste automatizado deve ser reavaliado, ele primeiro é executado e se e somente se o resultado deste teste retornar “sem sucesso” que será dada atenção a ele. A partir deste ponto, o automatizador de teste da equipe de teste automatizado irá iniciar uma investigação do motivo pelo qual tal teste automatizado retornou sem sucesso. A partir desta investigação que ele saberá se deve ou não refatorar o teste automatizado por conta de uma mudança na regra da documentação que ele abrange.

O ponto mais falho que percebemos nesse processo de descoberta de quais testes automatizados reavaliar é a questão de um “falso positivo”. Um teste automatizado onde a regra que ele cobre sofreu alteração mas ainda retorne sucesso em sua execução, nunca será reavaliado.

9.2 Aplicação da ferramenta

Tendo o modo como são tratados os testes automatizados no Bridge, foi disponibilizado para alguns testadores (ágeis e da equipe de teste automatizado) a utilização da ferramenta em um cenário controlado para avaliação dos mesmos sobre a solução. Eles executaram a configuração inicial da ferramenta, definindo as *tags* para identificação do texto e simularam a mudança de texto de um requisito junto com a descoberta de qual teste automatizado deveria ser reavaliado.

Após a utilização da ferramenta foi pedido para que eles respondessem um questionário (apêndice 1) para que pudéssemos obter um resultado da percepção desses testadores sobre a solução proposta para o problema.

Pergunta 1: Você tem quanto tempo de experiência com teste automatizado?

Gráfico 1 - Pergunta 1

Tal pergunta foi elaborada com o intuito de conhecermos nossos entrevistados em termos de experiência prática com a automatização de testes.

Pergunta 2: Qual sua principal atribuição com testes automatizados?

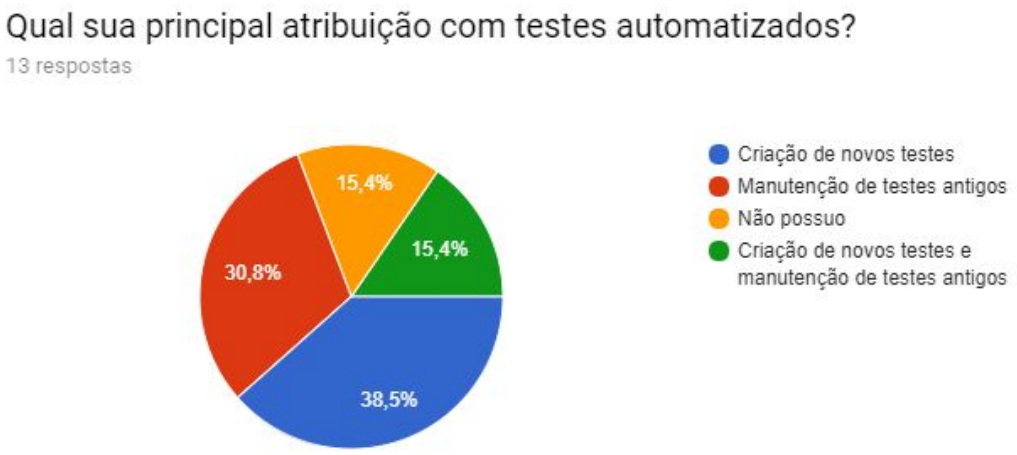

Gráfico 2 - Pergunta 2

A questão acima busca destacar, dentro da automatização de testes, qual função atribuída ao testador é a principal. Desta maneira, conheceremos, dentro de nosso universo de pesquisa, a função mais comum de um automatizador de testes.

Pergunta 3: Você costuma criar casos de teste automatizados baseados nos requisitos documentados?

Gráfico 3 - Pergunta 3

Como o problema identificado é justamente a inexistência de um rastreio entre a mudança de requisitos e casos de testes que derivam dos mesmos, elaboramos tal pergunta a fim de conhecermos se casos de teste são realmente desenvolvidos a partir dos requisitos documentados.

Pergunta 4: Quanto tempo você diria que dedica à manutenção/edição/atualização de casos de teste já criados?

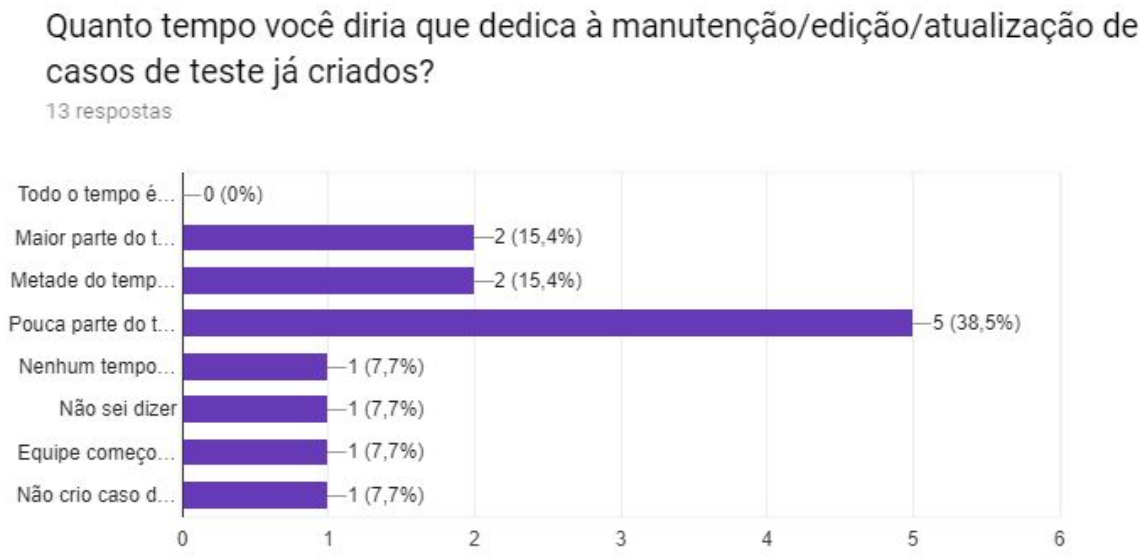

Gráfico 4 - Pergunta 4

Este questionamento visa descobrir se a alteração de casos de teste é uma tarefa presente na jornada de trabalho do testador, uma vez que a solução proposta pretende auxiliar justamente essa função.

Pergunta 5: Com o uso da ferramenta proposta, o tempo levado para alterar o caso de teste:

Gráfico 5 - Pergunta 5

Desta maneira será possível compreender se a ferramenta atingiu um de seus objetivos: auxiliar na rastreabilidade de mudança de requisitos e seus casos de teste automatizados. Também será possível quantificar a performance da solução.

Pergunta 6: Em uma escala de 1 a 5, como você classificaria a complexidade de utilizar a ferramenta?

Gráfico 6 - Pergunta 6

O propósito desta questão é justamente ilustrar a facilidade ou dificuldade que os entrevistados tiveram em manipular a solução e seu fluxo de uso proposto. Sendo 1, para uma classificação de uso menos complexa, escalável até 5, mais complexa.

Pergunta 7: Ficou mais fácil identificar os CDTs que precisam ser revalidados?

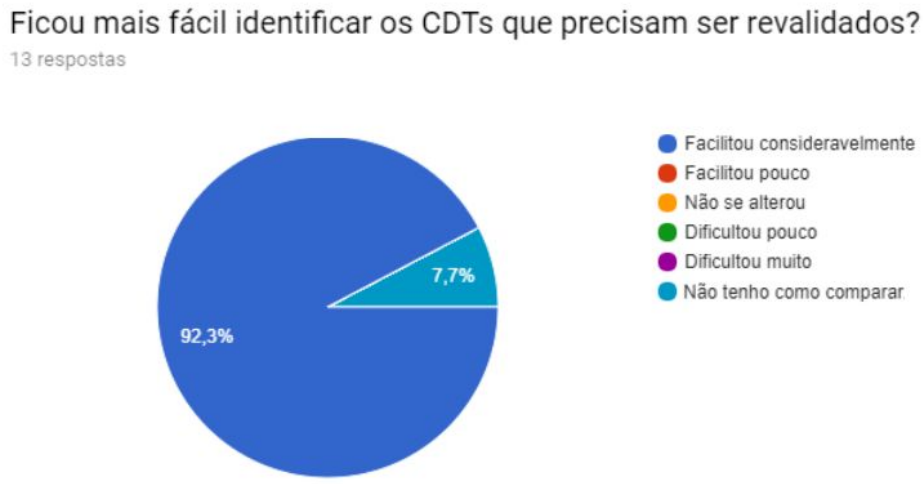

Gráfico 7 - Pergunta 7

A partir dos resultados deste questionamento, poderemos concluir e quantificar o êxito ou fracasso da solução proposta.

Pergunta 8: Você usaria a ferramenta proposta?

Gráfico 8 - Pergunta 8

Através das respostas obtidas será notável o real uso da ferramenta: se disponível para uso rotineiro, a mesma será ou não utilizada.

Pergunta 9: Alguma sugestão ou feedback que queira dar sobre a ferramenta?

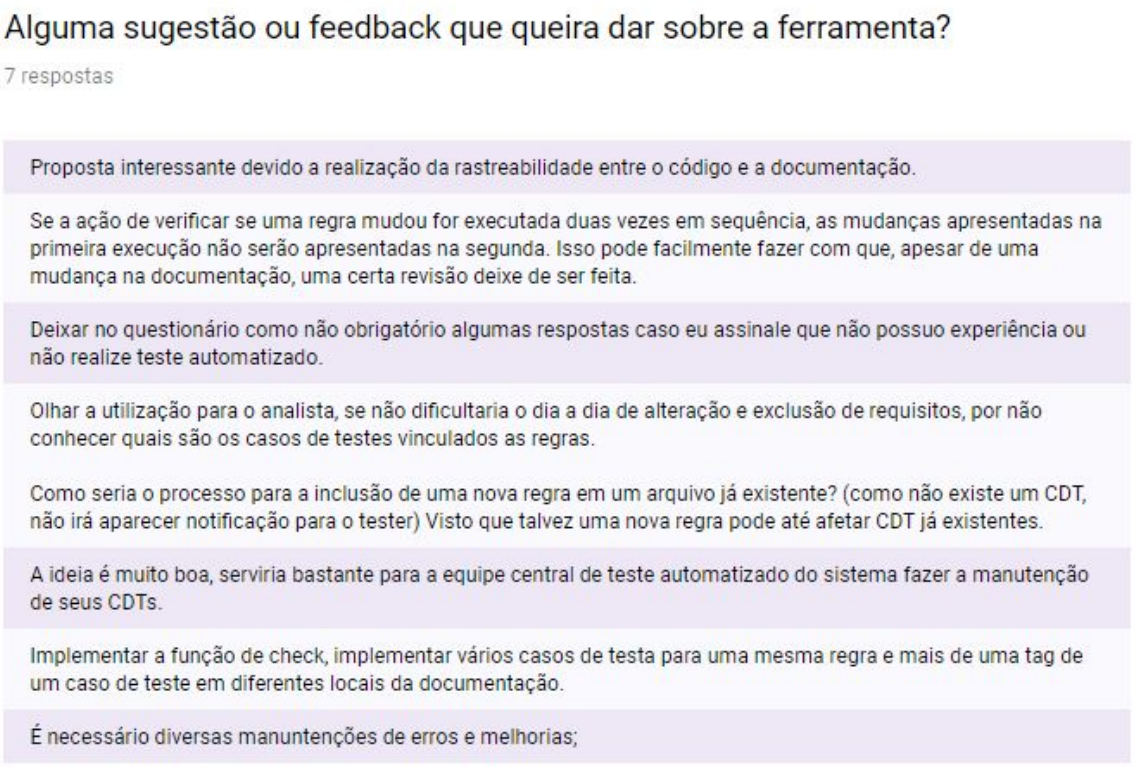

Figura 1 - Pergunta 9

Finalmente, fizemos tal pergunta com o intuito de colhermos a opinião dos entrevistados sobre a ferramenta desenvolvida, possíveis sugestões para trabalhos futuros e limitações encontradas durante seu uso.

A partir dos resultados obtidos na aplicação do questionário, com a pergunta número 1, podemos perceber que a maioria dos entrevistados possuem pouca experiência com teste automatizado, 38.5%, ou nenhuma experiência, 15,4%. Este resultado reflete que o teste automatizado é uma abordagem

difundida há pouco tempo entre os testadores do Laboratório Bridge, contudo, possuindo pelo menos 5 colaboradores entrevistados com mais de 1 ano de experiência.

A maioria destes testadores tem como sua principal atribuição a criação de novos testes automatizados, 38,5%, como é possível verificar pelas respostas obtidas com a pergunta número 2. Também é possível concluir que outra grande parcela tem como principal função a manutenção de testes automatizados, 30,8%. Nota-se que há mais pessoas criando testes do que prestando manutenção aos mesmos. A função conjunta de criação e manutenção é destinada a apenas 2 colaboradores, como visto na pergunta 1.

É possível verificar que todos que criam testes automatizados, utilizam a documentação do *software*, contendo seus requisitos, como guia para planejar seus testes, 76,9%. Os outros 23,1% não criam testes automatizados.

Vemos, também, que a maioria dos entrevistados, 38,5%, dedica pouco tempo à manutenção dos testes automatizados. Esse resultado pode ser explicado visto que a maioria dos entrevistados tem como principal função a criação destes testes, como apontado nas perguntas anteriores. Porém, 4 entrevistados (30,8%) dedicam, no mínimo, metade do seu tempo à manutenção/edição/atualização de casos de teste já criados.

É possível perceber, pela pergunta 5, que 46,2% dos entrevistados acreditam que a ferramenta proposta diminui consideravelmente o tempo levado para encontrar a regra que sofreu alguma mudança e qual caso de teste deve ser alterado. Nenhum testador relatou que houve aumento no tempo e 23,1% não

souberam dizer porque gostariam de ter usado a ferramenta em sua rotina de trabalho, não apenas no cenário em que a solução foi demonstrada.

Ao serem questionados a respeito da complexidade de utilização da ferramenta após a demonstração prática, nenhum participante julgou a solução como sendo muito alta (nível de complexidade 5) ou alta complexidade (nível de complexidade 4). Assim, podemos concluir que a ferramenta e seu fluxo de uso proposto são facilmente entendidos, uma vez que 53,8% concluiu como baixa complexidade (nível de complexidade 2) e 30,8% como sendo muito baixa (nível de complexidade 1).

Após o uso da ferramenta no cenário proposto, a grande maioria dos entrevistados, 92,3% acredita que a tarefa de encontrar o caso de teste automatizado a ser reavaliado ficou consideravelmente mais fácil. Vale ressaltar que o único voto diferente dos demais foi registrado por um participante que gostaria de utilizar o cenário proposto, também, sem a implementação da ferramenta, para uma melhor quantificação de sua performance.

Por fim, ao serem questionados se usariam a ferramenta proposta em suas rotinas, 76,9% afirmaram que usariam a solução da maneira que lhes foi demonstrada. Apenas um participante não usaria a ferramenta pois gostaria que a mesma fosse implementada em forma de *plugin* ou com interface gráfica. Tal sugestão foi indicada como possível trabalho futuro.

10 CONCLUSÃO E TRABALHOS FUTUROS

10.1 Conclusão

Com o intuito de desenvolver e aplicar uma solução para a inexistência de um rastreio entre a mudança de requisitos e casos de testes que derivam dos mesmos, este trabalho objetivou apresentar uma ferramenta que automatize tal rastreio.

Visando este objetivo, foi elaborado e desenvolvido um experimento, aplicando a ferramenta desenvolvida em um cenário controlado dentro do Laboratório Bridge. A fim de avaliar diversos pontos da solução proposta como: usabilidade, eficácia e avaliação da existência do problema, um questionário foi elaborado e aplicado a automatizadores e testadores ágeis do Laboratório.

Por meio da avaliação aplicada aos 13 entrevistados, algumas conclusões obtiveram destaque:

- Os entrevistados mostraram ter um tempo de experiência heterogêneo, sendo maior parte menos de 6 meses e a segunda maior parte sendo mais de 2 anos de experiência;
- Criação de novos testes ou manutenção de testes antigos demonstrou ser a atribuição mais frequente dos testadores;

- A grande maioria (76,9%) costuma criar casos de teste automatizados baseados em requisitos documentados, demonstrando assim, que uma ferramenta que auxilie este cenário terá grande aceitação;
- 77% dos entrevistados constataram que, com o uso da ferramenta, o tempo levado para identificar o caso de teste a ser alterado diminui de alguma forma.
- Não houveram avaliações nível 4 ou 5 (mais complexo) de complexidade no uso da ferramenta. 84,6% dos entrevistados avaliaram como nível 1 ou 2 (menos complexo).
- Dos 13 entrevistados, 12 (92,3%) constataram que a ferramenta facilitou consideravelmente a identificação de casos de testes a serem revalidados.
- 10 entrevistados (76,9%) usariam a ferramenta proposta em sua rotina de trabalho. Sendo que, apenas 1 (7,7%) passaria a usar se melhorias fossem implementadas.

Este trabalho alcançou seus objetivos em propor, desenvolver, aplicar e analisar uma solução para o problema apresentado. Com as respostas obtidas no questionário, é também possível concluir que a solução obteve êxito em sua principal funcionalidade e pode ser utilizada na rotina dos testadores, facilitando sua tarefa de verificar se mudanças em requisitos documentados, devem gerar ou não, atualização nos casos de teste automatizados derivados dos mesmos.

Devido à grande aceitação por parte dos entrevistados, como é visível no questionário, melhorias serão aplicadas nos pontos fracos e nos problemas conhecidos da ferramenta para que a mesma passe a fazer parte ativa da rotina

de trabalho dos testadores do Bridge. Tais melhorias são sugeridas na próxima seção. Os pontos fortes da solução proposta não necessitarão de alteração, como por exemplo as três operações (*check, trace* e *verify)* e usabilidade da ferramenta em si.

10.2 Trabalhos futuros

Como sugestão de trabalhos futuros deixamos esta seção descrita de problemas encontrados, apontando:

- A implementação da funcionalidade de rastrear um teste automatizado que cubra mais de uma regra.
- Implementação de um arquivo de propriedade, fechando o código da solução em um *.jar,* para assim facilitar o uso da ferramenta.
- Implementar a rastreabilidade de adição de novos textos à documentação.
- Implementar a rastreabilidade de exclusão de regras da documentação, com suas *tags.*
- Implementar uma interface gráfica para melhor controle da operação número três, *check.*
- Implementar uma validação para as *tags,* descritas no texto da documentação.

- Implementar uma validação para classes de testes automatizados com nomes iguais, pegando o seu *path,* por exemplo.

Destacamos que a implementação da solução realizada neste trabalho de conclusão de curso foi focada para resolver o problema encontrado no Laboratório Bridge. Contudo, salientamos que a ideia da solução proposta neste trabalho pode ser aplicada de uma forma mais abrangente a quaisquer cenários em geral onde necessita-se de uma rastreabilidade de texto para texto, como feito. Uma implementação diferente ou ajustes serão necessários em questão de código.

REFERÊNCIAS BIBLIOGRÁFICAS

1. S. LLIEVA; S. UNIV; P. IVANOV. Analyses of an agile methodology implementation. IEEE: Euromicro Conference, 2004. Proceedings, v. 30, p. 326-333, 2004.

2. V. RAHIMIAN; R. RAMSIN; P. IVANOV. Designing an agile methodology for mobile software development: A hybrid method engineering approach. IEEE: 2008 Second International Conference on Research Challenges in Information Science. Proceedings, p. 337-342, 2008.

3. L. MARUPING; V. VENKATESH; R. AGARWAL. A Control Theory Perspective on Agile Methodology Use and Changing User Requirements. INFORMS, p. 377-399, 2009.

4. Kurt Wallnau, Scott Hissam, e Robert Seacord. Building Systems from Commercial Components.SEI Series in Software Engineering.Addison-Wesley 2002.

5. PRESSMAN, Roger S. Engenharia de Software. Sexta edição. São Paulo: McGraw-Hill, 2006.

6. PRESSMAN, Roger S. Engenharia de Software. Terceira edição. São Paulo: Pearson Makron Books, 1995.

7. P. Naur e B. Randall. Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee. The Scientific Affairs Committee, NATO. Bruxelas. 1969.

8. Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Roland Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile *Software* Development, 2001.

9. Carlos Renato Rocha Bueno, Regina Sélia de Almeida Rodrigues Bueno, Paulo Sergio Borba. Metodologias Ágeis – Extreme Programming e o Tratamento de Requisitos, 2008

10. Somerville, I. Engenharia de *software*. 6° ed. Tradução Maurício de Andrade. São Paulo: Ed Addison-Wesley, 2003

11. MYERS, Glenford J. (2004). The Art of *Software* Testing 2 ed. (Nova Jérsei: John Wiley & Sons), pg 8.

12. H.F.Hofmann,F. Lehner. “Requirements Engineering as a Success Factor in *Software* Projects, IEEE *Software*, July/August 2001.

13. SOARES, Michel dos Santos. Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de *Software*. INFOCOMP, [S.l.], v. 3, n. 2, p. 8-13, nov. 2004. ISSN 1982-3363. Available at: <http://infocomp.dcc.ufla.br/index.php/INFOCOMP/article/view/68>. Data de acesso: 12 out. 2017.

14. BERNARDO, Paulo Cheque; KON, Fabio. A importância dos Testes Automatizados. Engenharia de *Software* Magazine, v. 1, n. 3, p. 54-57, 2008.

15. Saswat Anand, Edmund K.Burke, Tsong Yueh Chen, John Clarkd, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn. “An

orchestrated survey of methodologies for automated *software* test case generation". Journal of Systems and *Software*, Volume 86, Issue 8, August 2013, Pages 1978-2001.

16. KOSCIANSKI, André; SOARES, Michel dos Santos. Qualidade de *software*. São Paulo: Novatec, v. 3, 2006.

17. AMBLER, Scott W. Agile Documentation. 2001-2004, The Official Agile Modeling (AM) Site, 2001, Disponível em: http://agilemodeling.com/essays/agileDocumentation.htm , Acesso em: 21 out. 2017.

18. Gotel, Orlena & Finkelstein, A. "An Analysis of the Requirements Traceability Problem". In: IEEE International Conference on Requirements Engineering, IEEE Computer Society Press, Colorado Springs, Colorado, 1994. Proceedings. pp. 94-101.

19. WAZLAWICK, Raul Sidnei "Análise e projeto de sistemas de informação orientados a objetos" – 2.ed. – Rio de Janeiro: Elsevier, 2011. (Série SBC, Sociedade Brasileira de Computação), p. 30-31.

20. BEZERRA, Eduardo, 1972 - "Princípios de análise e projeto de sistemas com UML" - 3. ed. - Rio de Janeiro: Elsevier, 2015, p. 48 - 58.

21. MÜLLER, T. et al. Base de Conhecimento para Certificação em Teste-Foundation Level Syllabus. ISTQB-Comissão Internacional para Qualificação de Teste de *Software*, 2007.

22. https://www.java.com/en/download/faq/whatis_java.xml acessado em outubro de 2017

23. https://daringfireball.net/projects/markdown/ acessado em outubro de 2017

24. https://www.caelum.com.br/apostila-java-orientacao-objetos/um-pouco-de-arrays/ acessado em outubro de 2017

25. http://www.devmedia.com.br/trabalhando-com-string-string-em-java/21737 acessado em outubro de 2017

26. http://www.devmedia.com.br/como-funciona-a-criptografia-hash-em-java/31139 acessado em outubro de 2017

27. DE PÁDUA PAULA FILHO, Wilson. Engenharia de *software*. LTC, 2003, p. 15

28. https://bridge.ufsc.br/ acessado em outubro de 2017

29. PMBOK, GUIDE. Um guia do conhecimento em gerenciamento de projetos. Quarta Edição, 2013.

30. Cruz, Jorge & Jino, Mario & Crespo, Adalberto & Argollo, Miguel. (2006). Suporte automatizado à rastreabilidade em um processo de teste de *software* baseado em documentação.

31. https://cucumber.io/ acessado em outubro de 2017.

32. IEEE (1998), The Institute of Electrical and Electronics Engineers, “IEEE Std 829: Standard for *Software* Test Documentation”, IEEE Computer Society, Setembro.

33. https://github.com/explore acessado em novembro de 2017.

34. http://www.seleniumhq.org/about/ acessado em novembro de 2017.

Apêndice 1 - Questionário

Questionário para avaliar a utilidade da ferramenta desenvolvida como parte do Trabalho de Conclusão de Curso dos alunos Antonio Donatti e Leonardo Augusto do curso de Sistemas de Informação da Universidade Federal de Santa Catarina e aplicado no laboratório Bridge da Universidade Federal de Santa Catarina.

Pergunta 1: Você tem quanto tempo de experiência com teste automatizado?

- Menos de 6 meses
- Entre 6 meses e 1 ano
- De 1 a 2 anos
- Mais de 2 anos
- Não possui experiência com testes automatizados

Pergunta 2: Qual sua principal atribuição com testes automatizados?

- Criação de novos testes
- Manutenção de testes antigos
- Criação de novos testes e manutenção de testes antigos
- Outros… (Comente)
- Não possuo

Pergunta 3: Você costuma criar casos de teste automatizados baseados nos requisitos documentados?

- Sim
- Não
- Não crio casos de teste
- Outros… (Comente)

Pergunta 4: Quanto tempo você diria que dedica à manutenção/edição/atualização de casos de teste já criados?

- Todo o tempo é destinado
- Maior parte do tempo é destinado
- Metade do tempo é destinado
- Pouca parte do tempo é destinado
- Nenhum tempo é destinado
- Não sei dizer
- Outros… (Comente)

Pergunta 5: Com o uso da ferramenta proposta, o tempo levado para alterar o caso de teste:

- Diminuiu consideravelmente
- Diminuiu um pouco
- Não se alterou
- Aumentou um pouco

- Aumentou consideravelmente
- Não sei dizer

Pergunta 6: Em uma escala de 1 a 5 (menos complexo para mais complexo), como você classificaria a complexidade de utilizar a ferramenta?

- 1
- 2
- 3
- 4
- 5

Pergunta 7: Ficou mais fácil identificar os CDTs que precisam ser revalidados?

- Facilitou consideravelmente
- Facilitou pouco
- Não se alterou
- Dificultou pouco
- Dificultou muito
- Outros… (Comente)

Pergunta 8: Você usaria a ferramenta proposta?

- Sim
- Não
- Não sei dizer
- Outros… (Comente)

Pergunta 9: Alguma sugestão ou feedback que queira dar sobre a ferramenta?

- Texto de resposta longa

Apêndice 2 - Código-fonte

2.1 - Classe “Agregador”

```
import java.util.ArrayList;

public class Agregador {

	public Agregador() {

	}

	public ArrayList<ObjTrack> agregar(ArrayList<String> doc,
ArrayList<String> selenium) {

		ArrayList<ObjTrack> arquivo = new ArrayList<ObjTrack>();

		for (int i = 0; i < doc.size(); i++) {
			for (int j = 0; j < doc.get(i).length(); j++) {
				if (doc.get(i).charAt(j) == '{') {
					String temp = doc.get(i).substring(0, j);
					String tagDoc = this.limpaString(temp);
```

```
                                        String hash = doc.get(i).substring(j + 1,
doc.get(i).length() - 1);

                                        ObjTrack novo = new ObjTrack(tagDoc, hash,
null);
                                        arquivo.add(novo);
                                }
                        }
                }

                for (int i = 0; i < selenium.size(); i++) {
                        for (int j = 0; j < selenium.get(i).length(); j++) {
                                if (selenium.get(i).charAt(j) == ',') {
                                        String tagSelen = selenium.get(i).substring(0, j);
                                        String classe = selenium.get(i).substring(j + 1,
selenium.get(i).length());

                                        for (int h = 0; h < arquivo.size(); h++) {
                                                if
(arquivo.get(h).getTag().equals(tagSelen)) {
                                                        arquivo.get(h).setClasse(classe);
                                                }
                                        }
```

```
                        }
                    }
                }
                return arquivo;
        }

        public String limpaString(String line) {
                line = line.trim();
                line = line.replace(" ", "");
                line = line.replace("<", "");
                line = line.replace(">", "");
                line = line.replace("-", "");
                line = line.replace("!", "");

                return line;
        }
}
```

2.2 - Classe “CalculaHash

```
import java.security.MessageDigest;

public class CalculaHash {
```

```
	public CalculaHash() {

	}

	public String getTag(String t) {
		int inicio = 0;
		int flag = -1;
		String tag = "";

		for (int i = 0; i > flag; i++) {
			if (t.charAt(i) == '-' && t.charAt(i + 1) == '-' && t.charAt(i + 2) ==
'>') {
				flag = i + 10;
				inicio = i + 3;
			}

		}

		tag = t.substring(0, inicio);

		return tag;
	}
```

```
        public String getTexto(String t) {
                String texto = "";
                int comeco = 0;
                int fim = 0;
                int flag = -1;

                for (int i = 0; i > flag; i++) {
                        if (t.charAt(i) == '-' && t.charAt(i + 1) == '-' && t.charAt(i + 2) ==
'>') {
                                flag = i + 10;
                                comeco = i + 3;

                        }
                }

                for (int i = 0; i < t.length(); i++) {
                        if (t.charAt(i) == 'F' && t.charAt(i + 1) == 'I' && t.charAt(i + 2)
== 'M') {
                                fim = i - 4;
                        }
                }
```

```
            texto = t.substring(comeco, fim);

            return texto;
        }

        public String stringComTagHash(String t) {
            String stringTagHash = "";
            String tag = this.getTag(t);
            String texto = this.getTexto(t);

            String hash = this.generateHash(texto);

            stringTagHash = tag + "{" + hash + "}";

            return stringTagHash;
        }

        public String generateHash(String texto) {

            try {
                MessageDigest algorithm =
MessageDigest.getInstance("SHA-256");
```

```
                byte messageDigest[] =
algorithm.digest(texto.getBytes("UTF-8"));

                StringBuilder hexString = new StringBuilder();
                for (byte b : messageDigest) {
                        hexString.append(String.format("%02X", 0xFF & b));
                }
                String textoHex = hexString.toString();
                return textoHex;
            } catch (Exception e) {
                e.printStackTrace(System.err);
                return null;
            }
        }
}
```

2.3 - Classe “Comparador”

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
```

```
public class Comparador {

	public Comparador() {

	}

	public void compararArquivos(File antigo, ArrayList<ObjTrack> novo)
throws IOException {

		ArrayList<ObjTrack> velho = this.montaArray(antigo);

		for (int i = 0; i < novo.size(); i++) {
			for (int j = 0; j < velho.size(); j++) {
				if (novo.get(i).getTag().equals(velho.get(j).getTag())) {
					if
(!novo.get(i).getHash().equals(velho.get(j).getHash())) {
						System.out.println("Rever Caso de Teste
" + novo.get(i).getClasse() + " pois a regra " + novo.get(i).getTag() + " mudou!");
					} else {
						System.out.println("Regra " +
novo.get(i).getTag() + " conserva-se inalterada!");
					}
```

```
                }
            }
        }

    }

    public ArrayList<ObjTrack> montaArray(File f) throws IOException {
        ArrayList<ObjTrack> velho = new ArrayList<ObjTrack>();
        BufferedReader br = new BufferedReader(new
InputStreamReader(new FileInputStream(f), "UTF-8"));

        String line;
        String tag = "";
        String hash = "";
        String classe = "";
        int fimTag = 0;

        while ((line = br.readLine()) != null) {
            for (int i = 0; i < line.length(); i++) {
                if (line.charAt(i) == '{') {
                    tag = line.substring(0, i);
                    fimTag = i;
                }
```

```
                        if (line.charAt(i) == '}') {
                                hash = line.substring(fimTag + 1, i);
                                classe = line.substring(i, line.length());
                        }
                }
                ObjTrack obj = new ObjTrack(tag, hash, classe);
                velho.add(obj);
        }
        br.close();

        return velho;
    }
}
```

2.4 - Classe “LeitorDoc”

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;

public class LeitorDoc {
```

```
public LeitorDoc() {

}

ListaDoc lista = new ListaDoc();

public ArrayList<String> leitor() throws IOException {
        ArrayList<String> lista = new ArrayList<String>();

        for (int i = 0; i < this.lista.listaDoc.length; i++) {
                File docFile = this.lista.listaDoc[i];

                BufferedReader br = new BufferedReader(new
InputStreamReader(new FileInputStream(docFile), "UTF-8"));

                String line;
                String tag = null;
                String concat = "";

                while ((line = br.readLine()) != null) {

                        if (line.contains("<!--RRCDT_")) {
```

```
                    if (tag == null) {
                        tag = line; // this.getTag(line);
                    }
                } else if (tag != null) {
                    concat = concat + line;
                }

                if (line.contains("<!--FIM")) {
                    tag = tag + concat;

                    CalculaHash calc = new CalculaHash();
                    String stringzao = calc.stringComTagHash(tag);

                    lista.add(stringzao);
                    concat = "";
                    tag = null;
                }
            }
            br.close();
        }
        return lista;
    }
```

```
}
```

2.5 - Classe “LeitoSelenium”

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;

public class LeitorSelenium {

	public LeitorSelenium() {

	}

	ListaSelenium lista = new ListaSelenium();

	public ArrayList<String> leitor() throws IOException {
		ArrayList<String> lista = new ArrayList<String>();

		for (int i = 0; i < this.lista.listaSele.length; i++) {
			File seleFile = this.lista.listaSele[i];
```

```
                    BufferedReader br = new BufferedReader(new
InputStreamReader(new FileInputStream(seleFile), "UTF-8"));

                    String line;
                    String tag = "";
                    String concat = "";
                    String classe = "";

                    while ((line = br.readLine()) != null) {

                        if (line.contains("@track")) {
                            tag = this.getTag(line);
                            classe = seleFile.getName();
                            concat = tag + "," + classe;
                            lista.add(concat);
                        }
                    }
                    br.close();
                }
                return lista;
            }
```

```
        public String getTag(String line) {
                for (int i = 0; i < line.length(); i++) {
                        if (line.charAt(i) == '@' && line.charAt(i + 1) == 't' &&
line.charAt(i + 2) == 'r' && line.charAt(i + 3) == 'a' && line.charAt(i + 4) == 'c'
                                        && line.charAt(i + 5) == 'k') {
                                line = line.substring(i + 6);
                                line = line.replace(" ", "");
                                line = line.trim();
                                return line;
                        }
                }

                return null;
        }

}
```

2.6 - Classe “ListaDoc”

```
import java.io.File;
import java.util.ArrayList;
import java.util.Vector;

public class ListaDoc {
```

```
	// File dirDoc = new File("C:/Users/leo14/Documents/Projeto/esus/esus/documentacao/_docs");
	File dirDoc = new File("C:/Users/leo14/Desktop/TCC/Exemplos");
	File[] listaDoc;

	public ListaDoc() {
		this.listaDoc = this.listDir(this.dirDoc);
	}

	public File[] listDir(File dir) {
		Vector enc = new Vector();
		File[] files = dir.listFiles();
		for (int i = 0; i < files.length; i++) {
			if (files[i].isDirectory()) {
				// Adiciona no Vector os arquivos encontrados dentro de 'files[i]':
				File[] recFiles = this.listDir(files[i]);
				for (int j = 0; j < recFiles.length; j++) {
					enc.addElement(recFiles[j]);
				}
			} else {
				// Adiciona no Vector o arquivo encontrado dentro de 'dir':
```

```
                    enc.addElement(files[i]);
                }
            }
            // Transforma um Vector em um File[]:
            File[] encontrados = new File[enc.size()];
            for (int i = 0; i < enc.size(); i++) {
                encontrados[i] = (File) enc.elementAt(i);
            }

            ArrayList<File> list = new ArrayList();
            for (int i = 0; i < encontrados.length; i++) {
                if (encontrados[i].getName().endsWith(".md")) {
                    list.add(encontrados[i]);
                }
            }
            File[] lista = new File[list.size()];

            for (int i = 0; i < list.size(); i++) {
                lista[i] = list.get(i);
            }
            return lista;
        }
```

```
}
```

2.7 - Classe “ListaSelenium”

```
import java.io.File;
import java.util.ArrayList;
import java.util.Vector;

public class ListaSelenium {

	// File dirDoc = new File("C:/Users/leo14/Documents/Projeto/esus/selenium");
	File dirSele = new File("C:/Users/leo14/Desktop/TCC/Exemplos");
	File[] listaSele;

	public ListaSelenium() {
		this.listaSele = this.listDir(this.dirSele);
	}

	public File[] listDir(File dir) {
		Vector enc = new Vector();
		File[] files = dir.listFiles();
		for (int i = 0; i < files.length; i++) {
			if (files[i].isDirectory()) {
```

```
                        // Adiciona no Vector os arquivos encontrados dentro
de 'files[i]':
                        File[] recFiles = this.listDir(files[i]);
                        for (int j = 0; j < recFiles.length; j++) {
                                enc.addElement(recFiles[j]);
                        }
                } else {
                        // Adiciona no Vector o arquivo encontrado dentro de
'dir':
                        enc.addElement(files[i]);
                }
        }
        // Transforma um Vector em um File[]:
        File[] encontrados = new File[enc.size()];
        for (int i = 0; i < enc.size(); i++) {
                encontrados[i] = (File) enc.elementAt(i);
        }

        ArrayList<File> list = new ArrayList();
        for (int i = 0; i < encontrados.length; i++) {
                if (encontrados[i].getName().startsWith("CDT")) {
                        list.add(encontrados[i]);
                }
```

```
                }
                File[] lista = new File[list.size()];

                for (int i = 0; i < list.size(); i++) {
                        lista[i] = list.get(i);
                }
                return lista;
        }

}
```

2.8 - Classe “main”

```
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;

public class Main {

        public static void main(String arg[]) throws IOException {

                LeitorDoc leitorDoc = new LeitorDoc();
                ArrayList<String> arrayDoc = leitorDoc.leitor();
```

```
                LeitorSelenium leitorSelen = new LeitorSelenium();
                ArrayList<String> arraySelen = leitorSelen.leitor();

                Agregador junta = new Agregador();

                ArrayList<ObjTrack> arquivo = junta.agregar(arrayDoc, arraySelen);

                Persistir p = new Persistir();

                Comparador c = new Comparador();

                File bd = new
File("C:/Users/leo14/Desktop/TCC/Exemplos/BD_Regras.txt");
                if (bd.exists()) {
                        c.compararArquivos(bd, arquivo);
                        bd.delete();
                }
                p.escreveArquivo(arquivo);

        }


}
```

2.9 - Classe “ObjTrack”

```
public class ObjTrack {

	String tag;
	String hash;
	String classe;

	public ObjTrack(String t, String h, String c) {
		this.tag = t;
		this.hash = h;
		this.classe = c;
	}

	@Override
	public String toString() {
		String tudo = this.tag + "{" + this.hash + "}" + this.classe;
		return tudo;
	}

	public String getTag() {
```

```
        return this.tag;
}

public void setTag(String tag) {
        this.tag = tag;
}

public String getHash() {
        return this.hash;
}

public void setHash(String hash) {
        this.hash = hash;
}

public String getClasse() {
        return this.classe;
}

public void setClasse(String classe) {
        this.classe = classe;
}
```

```
}
```

2.10 - Classe “Persistir”

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;

public class Persistir {

	public Persistir() {

	}

	public void escreveArquivo(ArrayList<ObjTrack> array) throws IOException
{
		FileWriter arq = new
FileWriter("C:/Users/leo14/Desktop/TCC/Exemplos/BD_Regras.txt");
		PrintWriter gravarArq = new PrintWriter(arq);

		for (int i = 0; i < array.size(); i++) {
			gravarArq.printf(array.get(i).toString() + "%n");
```

```
                }


                arq.close();
                System.out.println("Arquivo de rastreabilidade criado com
sucesso!");
        }
}
```

Apêndice 3 - Artigo

Proposta de Rastreabilidade Entre Mudanças de Requisitos e seus Casos de Testes Automatizados

Antonio de Azevedo Donatti[1] , Leonardo Augusto da Silva Veiga[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 – 88.010-970 – Florianópolis – SC – Brasil

leo14s@gmail.com, antonioadonatti@gmail.com

Abstract. *The requirements documentation is the main artifact provided by the requirements engineering. Yet, inevitably, this documentation can be changed, impacting several stages of the software development. In a development environment, where automated testing is based on those documented requirements, such changes must be tracked visibly and responsibly, in order that the maintenance of automated test is the least costly possible. Therefore, the present work develops and evaluate a tool to serve the traceability between documented requirements and their automated test cases.*

Resumo. *A documentação dos requisitos é o principal artefato oriundo da engenharia de requisitos. Contudo, inevitavelmente, esta documentação poderá sofrer mudanças que irão impactar em diversos estágios do desenvolvimento do software. Em um ambiente de desenvolvimento onde os testes automatizados são elaborados de acordo com os requisitos documentados, tais mudanças devem ser rastreadas de forma visível e ágil, a fim de que a manutenção dos testes automatizados seja a menos custosa possível. Sendo assim, o presente artigo desenvolve e avalia uma ferramenta para o auxílio da rastreabilidade entre requisitos documentados e seus casos de testes automatizados.*

1. Introdução

A deficiência no tratamento de requisitos tem sido apontada como a principal causa de fracassos de projetos de software [1]. Tal tratamento pode ter sido gerado devido a uma mudança no requisito. Para um ambiente de desenvolvimento de software ser considerado ágil, o mesmo deve aceitar a mudança e adaptar-se a ela. O problema não é a mudança em si, mesmo porque ela ocorrerá de qualquer forma. O problema é como receber, avaliar e responder às mudanças [2]. Os testes automatizados verificam automaticamente funcionalidades do sistema e registram efeitos colaterais obtidos. Esta abordagem permite que todos os casos de teste sejam facilmente e rapidamente repetidos a qualquer momento e com pouco esforço [3]. Contudo, casos de teste automatizados que estejam associados a requisitos do software, também estarão sujeitos às mesmas mudanças. Como a parte de teste chega a custar até 50% do tempo de desenvolvimento, todo o tempo poupado nesta

etapa é válido [4]. Sendo assim, o presente trabalho visa propor e avaliar uma ferramenta que auxilie o automatizador a identificar estas mudanças para que seus casos de teste sejam revalidados.

1.2. Motivação e justificativa

Durante o aprendizado no Laboratório Bridge, foi identificado uma lacuna que faltava ser preenchida entre duas áreas do desenvolvimento de software: a rastreabilidade entre a documentação do software, suas especificações, produzida pela área da análise, e os casos de teste, produzidos pelo setor de qualidade. Este problema tem um impacto maior nos casos de testes produzidos pelos testes automatizados. Muitas vezes, com uma troca de regra de negócio gerada por um requisito, muito casos de teste automatizados acabavam "quebrando" por não se aplicarem mais ou porque sua especificação mudou, gerando muito retrabalho. Todos os casos de testes deveriam ser revistos por não se saber em quais casos aquela alteração impactaria. Buscamos uma solução para facilitar essa rastreabilidade entre as alterações feitas na documentação e seus casos de testes relacionados, para uma manutenção menor, mais rápida, simples e eficaz.

1.3. Objetivos

Esta seção tratará dos objetivos e metas a serem alcançados através da pesquisa e execução do trabalho de conclusão de curso.

1.3.1. Objetivos gerais

Esta pesquisa tem como objetivo propor, aplicar e validar as respostas obtidas no questionário avaliativo a respeito da nova ferramenta, desenvolvida para resolver os problemas relacionados à manutenção dos casos de teste automatizados vinculados aos requisitos do software.

1.3.2. Objetivos específicos

Além disso, esta pesquisa tem como objetivo estudar todos os conceitos relacionados a este assunto, englobando metodologia ágil, análise de requisitos e teste de software. Como objetivos específicos para este trabalho pode-se citar:

- Foram propostas operações mínimas que satisfaçam a resolução do problema identificado no laboratório Bridge.
- A partir das operações mínimas propostas, foi desenvolvida uma ferramenta que busque auxiliar na rastreabilidade entre requisitos documentados e seus casos de testes automatizados.
- Com a ferramenta desenvolvida, os autores aplicaram a solução proposta aos testadores do Laboratório Bridge.
- Após a aplicação da solução, um questionário foi realizado com os testadores a fim de extrair conclusões sobre a ferramenta, tais como: usabilidade, utilidade e êxito ou fracasso no cumprimento de sua principal função.

1.4. Organização do artigo

O projeto de pesquisa foi realizado em parceria com o Laboratório Bridge da Universidade Federal de Santa Catarina. O escopo do artigo foi definido de acordo com a especificação e regras definidas pela Biblioteca Universitária. O artigo é composto por uma introdução, que explica detalhadamente a motivação por trás do tema escolhido e os objetivos gerais e específicos da pesquisa realizada. Contêm também seções de conceitos básicos relacionados à área da aplicação e que, portanto, são explicados a qualquer usuário (leitor) do trabalho e do processo que foi desenvolvido. O artigo é uma pesquisa e, portanto, busca contribuir para a área científica. A seção de desenvolvimento explica isso através da elaboração de uma solução visando a rastreabilidade entre testes automatizados e a documentação de requisitos do software. Após o desenvolvimento destas seções, é apresentada uma conclusão do que foi aprendido durante a pesquisa e os anos de graduação dos autores e uma proposta para trabalhos futuros.

2. Conceitos Básicos

2.1. Análise de requisitos

O levantamento de requisitos é o processo de descobrir quais são as funções que o sistema deve realizar e quais são as restrições que existem sobre essas funções. Operações que venham a constituir a funcionalidade do sistema são chamadas de requisitos funcionais. As restrições sobre essas operações são conhecidas como requisitos não funcionais [5].

O início para toda a atividade de desenvolvimento de software é o levantamento de requisitos, sendo esta atividade repetida em todas as demais etapas da engenharia de requisitos.

2.1. Teste de software

Teste de Software é um conjunto de atividades antecipadamente planejadas e executadas sistematicamente para garantir a qualidade de um software [6]. Definindo um roteiro de teste, passos para execução, com estratégias e técnicas diferenciadas, conseguimos garantir que o produto atingiu vários aspectos especificados como: segurança, desempenho e confiabilidade; e executou corretamente para o propósito que foi projetado. Como nem sempre é possível deixar o software livre de defeitos [7], ele tem como objetivo expor o maior número de defeitos que o mesmo possui, para que possam ser tratadas em versões futuras.

2.1. Teste automatizado

Para Testes Automatizados, são utilizadas ferramentas de software para a elaboração, execução e controle dos testes. A partir de testes de software já consolidados, podemos automatizá-los, deixando a cargo da ferramenta, e não do usuário, a execução e verificação dos mesmos, facilitando a execução repetitiva do mesmo teste várias vezes. Como a execução dos testes de software pode ser um trabalho maçante, a escrita de testes automatizados permite que uma vasta gama de casos de testes podem ser validados rapidamente. A implantação de testes automatizados pode ser custosa no início, mas a

longo prazo, para software de grande porte ou que estão em produção durante muito tempo, economiza muito tempo de teste, principalmente em testes de regressão. Para a automação de testes, duas abordagens podem ser utilizadas: testes baseados na interface gráfica do usuário e testes baseados em código. Para testes baseados na interface gráfica do usuário, uma ferramenta de software simula as entradas no software que se quer testar como um usuário faria. Essas entradas são definidas pelo programador do teste, com base nos artefatos criados previamente, como os casos de teste, assim observando as ações esperadas. Em nível de código, o programador do teste constrói o teste direto no código, possibilitando os testes de unidade, que foca em seções do código do software testado, observando se estão executando da forma esperada.

3. Problemas existentes

Durante o tempo de estágio no Laboratório Bridge, identificamos um problema recorrente que dificultava o trabalho dos testadores: a revalidação de seus casos de teste quando ocorria uma alteração na documentação do software. Tal problema se amplifica quando se trata dos testes automatizados: por serem muitos testes, por não sabermos especificamente qual teste engloba qual regra, por não sabermos qual regra mudou e se essa mudança foi significativa a ponto de que uma refatoração do código do teste automatizado seja necessária.

Para o automatizador de teste, cujo testes automatizados são escritos baseados nas regras do software descritos na documentação, no caso do Laboratório Bridge, a incerteza nas mudanças das regras dificulta o processo de refatoração dos testes automatizados. Uma refatoração preventiva é inviabilizada, visto que o automatizador só saberá da mudança na regra quando o teste automatizado referente aquela regra for executado e retornar “sem sucesso”. No cenário de testes automatizados do Laboratório, um teste automatizado retornará “sucesso” na sua execução quando não houver problemas nas verificações feitas pelo mesmo, e retornará “sem sucesso” quando encontrar um problema nas verificações implementadas em seu código. Um problema maior é quando o teste automatizado de uma regra que sofreu alteração retornar “sucesso”. Neste caso, o teste automatizado se torna insuficiente, não testando o que ele deveria abranger da regra, deixando o processo de qualidade do software defasado por não garantir que aquela regra foi corretamente implementada no software. O automatizador de teste não saberá que tal teste automatizado está defasado por sempre retornar “sucesso” na sua execução.

Esse processo de refatoração dos testes sob demanda é considerado ineficiente levando em conta o porte dos software desenvolvidos pelo Laboratório. São necessários muitos testes automatizados para garantir a qualidade do produto. Como descrito anteriormente, os testes são baseados nas regras do software e tais regras estão sujeitas a mudanças constantes, gerando uma grande demanda na refatoração dos testes automatizados para a equipe responsável. É um processo custoso para o automatizador descobrir manualmente quais regras foram alteradas e quais testes automatizados devem ser revalidados pois, a mudança de regras só será visível utilizando a ferramenta git[13], onde a documentação do software está hospedada. No Laboratório Bridge, o código dos testes automatizados está organizado na forma de pacotes. Cada pacote é referente a um módulo do software que ele testa, dentro dele encontram-se as classes de testes automatizados. Para descobrir qual teste automatizado abrange determinada regra, o automatizador deverá navegar na estrutura de

pacotes para achar qual teste automatizado refere-se à regra desejada. Percebe-se que é um processo demorado, complexo, exaustivo, repetitivo e ineficiente, por isso não é feito no Laboratório. Pelos motivos apresentados, desenvolvemos uma ferramenta que facilita o processo de rastreabilidade.

4. Soluções possíveis

Analisando a ferramenta Matriz de Rastreabilidade [8], percebemos que é uma ferramenta que supre a necessidade de rastreabilidade, porém não provê o suporte automatizado necessário para uma descoberta ágil do que mudou na documentação do software e o que deve ser mudado nos testes automatizados. Servindo mais como um plano de cobertura de testes, para que o testador consiga rastrear os pontos que foram cobertos por testes.

Analisando a ferramenta Prometeu [9], percebemos que é uma ferramenta completa e complexa, implementando 8 tipos de documentos de teste e artefatos produzidos para a atividade de teste de software. Desta forma, a adoção de tal solução necessita de uma mudança no fluxo, na metodologia e na interação do ambiente de trabalho, sendo custoso. Baseada na norma IEEE-Std-829 [10], ela implementa uma série de documentos que dão suporte ao teste de software. Por ser muito complexa, os documentos produzidos pelo teste de software ficam todos centrados nela, impactando em uma mudança muito radical do processo de teste e análise de todo o laboratório. Procuramos uma ferramenta que se encaixe na realidade do problema. Toda a documentação estaria hospedada nela, mudando o fluxo de trabalho dos analistas. Com a solução proposta, o trabalho dos analistas não sofrerá um grande impacto pois tudo será feito pelo automatizador de testes.

A ferramenta Cucumber [10] usa um modelo diferente de teste abordado no Bridge, local onde evidenciamos o problema. Ela é baseada em BDD, que utiliza uma linguagem com sintaxe de descrição de comportamento em texto plano. Facilita para que todas as pessoas envolvidas no desenvolvimento de software entendam qual o comportamento do sistema, qual o comportamento do teste automatizado e o que o desenvolvedor deve implementar, pois a partir do texto documentado, o teste é escrito e, sobre o mesmo é implementada a funcionalidade. A rastreabilidade das funcionalidades com o teste será suprida, porém não temos o suporte automatizado das mudanças que desejamos.

5. Projeto e desenvolvimento da proposta

Pensando sobre como poderíamos solucionar esse problema, identificamos três operações que são necessárias para que a solução satisfaça nossas necessidades:

Obter uma ligação entre o texto da documentação e o código do teste automatizado, que chamamos de trace ou “rastro”; Identificar se ocorreu uma mudança no texto da documentação, que chamamos de verify ou “verificação”; Validar se a mudança identificada do texto impactou no teste e se ele deve ser refatorado, que chamamos de “check” ou “validar”.

A operação número um, trace, foi automatizada através da implementação de “tags”, códigos de identificação únicos, que representam um texto da documentação. Essa mesma tag deve ser incluida no código do teste automatizado, na forma de um comentário, para que o link seja feito. A operação número dois, verify, foi automatizada comparando o texto

da regra em um estado inicial, com o texto da mesma regra oriundo de uma mudança. Assim, unindo a operação um e dois, conseguimos identificar e mostrar ao automatizador de testes qual regra mudou e qual teste automatizado deve ser reavaliado. A operação número três, check, deve ser uma operação humana. Com o resultado das operações um e dois, o universo de testes automatizados e regras de documentação são reduzidos a somente aqueles que devem ser reavaliados. Assim, o automatizador de testes pode reavaliar os testes automatizados um por um, refatorando o que for necessário e validando quais as regras sofreram alteração mas não geram impacto no código do respectivo teste.

O desenvolvimento foi focado na solução do problema encontrado no Laboratório Bridge, de forma que a ferramenta interfira o mínimo possível no trabalho de outras pessoas que não são automatizadores de testes e no dos mesmo. A prévia utilização de outras ferramentas pelas partes interessadas, análise de software e qualidade de software, possibilitou uma abordagem para a codificação, descrita abaixo.

A execução da ferramenta foi dividida em três etapas. A preparação para que a rastreabilidade seja feita, o estabelecimento do estado dos requisitos, e a comparação dos estados dos requisitos. Os estados dos requisitos são definidos como estado alterado ou não alterado.

Com o conhecimento adquirido durante as disciplinas de programação cursadas na graduação, foi possível desenvolver a solução codificada em linguagem Java [12]. Analisando como poderíamos usar os artifícios da linguagem para codificar a solução, optamos por uma manipulação de arquivos.

A preparação começa com a definição das tags e sua inserção na documentação e no código das classes de testes automatizados. O usuário da ferramenta também deve informar em qual diretório do computador estão armazenadas as classes de testes automatizados e os arquivos da documentação do software.

Os artefatos produzidos pela análise de requisitos do Laboratório Bridge estão todos descritos em texto com markdown [11], possibilitando que sejam inseridos comentários no texto com as tags para gerar o identificador único da regra, no começo e no fim do trecho de texto que se deseja referenciar. Como são comentários e estão escritos em markdown, sua presença não impactará na leitura e utilização das outras pessoas que consomem esses artefatos, pois o comentário não aparecerá para o leitor. Sugerimos que para uma melhor utilização, cada trecho delimitado e definido pelas tag devem corresponder a um requisito do software ou parte do requisito.

As classes de código de teste automatizados do Laboratório Bridge são código Java com a utilização do framework Selenium [14]. Estas classes também podem ser tratadas pelo Java como um simples arquivo de texto, assim também possibilitando a inserção da tag em forma de comentário. Os comentários foram escolhidos como solução por não gerarem impacto na execução do código e na leitura dos requisitos do software documentado.

Com a inserção das tags nos arquivos da documentação e nos arquivos das classes de testes automatizados, a primeira etapa da execução da ferramenta, a preparação para o rastreio, se encerra. A segunda etapa, de estabelecimento do estado dos requisitos, começa a ser descrita abaixo. Com esses comentários, nos arquivos da documentação e arquivos das classes dos testes automatizados, e a manipulação de arquivos do Java, a solução

codificada consegue ler todos os arquivos da documentação do software, a partir de um diretório especificado e procurar pelos trechos de texto marcados com as tags. Identificando as tags, os trechos de texto são armazenados em um array de Strings para manipulação futura. Os trechos de texto de requisitos de cada tag marcada na documentação armazenados no array, então são transformados em um hash codificado em SHA256 [12] como artifício computacional para diminuir o tamanho do texto armazenado e para facilitar a comparação futura. Apenas o texto da documentação é transformado em hash. Do mesmo modo que são tratados os arquivos da documentação, a solução consegue ler todos os arquivos das classes de código dos testes automatizados, linha por linha, procurando e identificando as tags nos comentários, armazenando o nome da classe do teste automatizado, referente a cada tag, em um array de Strings. Com isso, a solução une, através da tag que está nos arquivos de testes automatizados e na documentação, em um único arquivo de texto, a tag identificadora, o hash do trecho de texto correspondente ao requisito, e o nome da classe do código do teste automatizado em uma única linha. Cada linha do arquivo de texto contém uma tag, um hash e o nome de uma classe. Este arquivo contém o estado atual da rastreabilidade das regras da documentação com seus respectivos testes automatizados. Assim é encerrada a segunda etapa de execução da ferramenta, que mapeia cada regra definida pelas tags na etapa anterior, com a classes de testes automatizados correspondentes, também identificadas pelas tags anteriormente. A terceira etapa de execução inicia-se quando houver a necessidade de identificar se houve uma mudança em um requisito rastreado. No Laboratório Bridge, o processo de atualização da documentação do software nunca deixou muito claro, explícito, o que foi alterado de uma versão para outra. Para o automatizador de teste, que tem seus testes automatizados escritos baseados na documentação, essa dúvida pode ser sanada somente acessando a ferramenta git [13], e comparando as versões através das utilidades da ferramenta, conseguindo ver o que foi alterado. Isto é um processo custoso para o automatizador pois, como há muitos analistas produzindo e alterando regras, muitas regras podem ter sido alteradas. Como explicado no capítulo anterior, esse processo de descoberta das mudanças geradas não é feito. Ao concluir as etapas um e dois, temos um arquivo de texto contendo a rastreabilidade da documentação com o teste automatizado, explicado anteriormente. Com o hash do texto da documentação feito na etapa dois, temos o estado atual da regra rastreada. No Laboratório Bridge, o uso da ferramenta git facilita o versionamento e trabalho dos analistas por possibilitar que todos alterem os mesmos arquivos simultaneamente, assim quando há uma alteração feita por algum analista, todos terão acesso a essas mudanças em seus arquivos. Do mesmo modo, os automatizadores de teste, que utilizam a documentação produzida pelos analistas, recebem as mudanças. Para identificar se houve uma mudança, primeiramente o automatizador de testes deve receber as mudanças vindas do git. A solução irá gerar novamente um arquivo de estado das regras, como descrito anteriormente, porém usará os arquivos modificados. Sabendo que as tags não foram alteradas de uma versão para outra, como explicado posteriormente na seção de problemas encontrados, um arquivo novo de estado das regras alteradas será gerado, como vimos. Estando em posse do arquivo de estado das regras e do arquivo de estado das regras alteradas, a solução irá comparar tag por tag se o hash do trecho de texto, referente a uma regra e identificado por uma tag, está diferente. Se o hash do estado da regra for diferente do estado da regra alterada, isso implica que o trecho de texto rastreado na documentação pelas tags sofreu alteração. A solução então irá listar todas as tags cujo

hash referente foi diferente nos arquivos comparados, exibindo para o automatizador de testes o nome da classe do teste automatizado referente a regra da documentação que ele abrange, referenciado pela tag. O arquivo de estado das regras alteradas passa a ser tratado como o estado atual das regras, e é armazenado para futuramente ser comparado novamente com um novo arquivo de estado das regras alteradas, finalizando o processo de execução da solução.

Sendo assim, a solução irá poupar o automatizador de teste de todo o trabalho necessário para descobrir previamente quais testes automatizados devem ser revalidados por causa de uma modificação em sua regra, facilitando, simplificando e encurtando o processo que vimos anteriormente. O grande impacto no processo do automatizador de teste será implantar as tags no primeiro momento e posteriormente mantê-las quando forem criadas novas regras na documentação que se deseja testar automatizadamente. Para o analista de software, em seu processo de trabalho ele deverá zelar pela integridade das tags inseridas pelo automatizador, assim entrando em comum acordo para que os dois elaborem um bom trabalho sem interferir no trabalho do seu colega.

6. Conclusão e Trabalhos Futuros

Com o intuito de desenvolver e aplicar uma solução para a inexistência de um rastreio entre a mudança de requisitos e casos de testes que derivam dos mesmos, este artigo objetivou apresentar uma ferramenta que automatize tal rastreio. Visando este objetivo, foi elaborado e desenvolvido um experimento, aplicando a ferramenta desenvolvida em um cenário controlado dentro do Laboratório Bridge. A fim de avaliar diversos pontos da solução proposta como: usabilidade, eficácia e avaliação da existência do problema, um questionário foi elaborado e aplicado a automatizadores e testadores ágeis do Laboratório.

Por meio da avaliação aplicada aos 13 entrevistados, algumas conclusões obtiveram destaque:

Os entrevistados mostraram ter um tempo de experiência heterogêneo, sendo maior parte menos de 6 meses e a segunda maior parte sendo mais de 2 anos de experiência; Criação de novos testes ou manutenção de testes antigos demonstrou ser a atribuição mais frequente dos testadores; A grande maioria (76,9%) costuma criar casos de teste automatizados baseados em requisitos documentados, demonstrando assim, que uma ferramenta que auxilie este cenário terá grande aceitação; 77% dos entrevistados constataram que, com o uso da ferramenta, o tempo levado para identificar o caso de teste a ser alterado diminui de alguma forma. Não houveram avaliações nível 4 ou 5 (mais complexo) de complexidade no uso da ferramenta. 84,6% dos entrevistados avaliaram como nível 1 ou 2 (menos complexo). Dos 13 entrevistados, 12 (92,3%) constataram que a ferramenta facilitou consideravelmente a identificação de casos de testes a serem revalidados. Dez entrevistados (76,9%) usariam a ferramenta proposta em sua rotina de trabalho. Sendo que, apenas 1 (7,7%) passaria a usar se melhorias fossem implementadas.

Este artigo alcançou seus objetivos em propor, desenvolver, aplicar e analisar uma solução para o problema apresentado. Com as respostas obtidas no questionário, é também possível concluir que a solução obteve êxito em sua principal funcionalidade e pode ser utilizada na rotina dos testadores, facilitando sua tarefa de verificar se mudanças em requisitos documentados, devem gerar ou não, atualização nos casos de teste automatizados

derivados dos mesmos. Devido à grande aceitação por parte dos entrevistados, como é visível no questionário, melhorias serão aplicadas nos pontos fracos e nos problemas conhecidos da ferramenta para que a mesma passe a fazer parte ativa da rotina de trabalho dos testadores do Bridge. Tais melhorias são sugeridas na próxima seção. Os pontos fortes da solução proposta não necessitarão de alteração, como por exemplo as três operações (check, trace e verify) e usabilidade da ferramenta em si.

Como sugestão de trabalhos futuros deixamos esta seção descrita de problemas encontrados, apontando:

A implementação da funcionalidade de rastrear um teste automatizado que cubra mais de uma regra; Implementação de um arquivo de propriedade, fechando o código da solução em um .jar, para assim facilitar o uso da ferramenta; Implementar a rastreabilidade de adição de novos textos à documentação; Implementar a rastreabilidade de exclusão de regras da documentação, com suas tags; Implementar uma interface gráfica para melhor controle da operação número três, check; Implementar uma validação para as tags, descritas no texto da documentação; Implementar uma validação para classes de testes automatizados com nomes iguais, pegando o seu path, por exemplo.

Destacamos que a implementação da solução realizada neste artigo foi focada para resolver o problema encontrado no Laboratório Bridge. Contudo, salientamos que a ideia da solução proposta neste artigo pode ser aplicada de uma forma mais abrangente a quaisquer cenários em geral onde necessita-se de uma rastreabilidade de texto para texto, como feito. Uma implementação diferente ou ajustes serão necessários em questão de código.

References

1. H.F.Hofmann,F. Lehner. “Requirements Engineering as a Success Factor in *Software* Projects, IEEE *Software*, July/August 2001.

2. SOARES, Michel dos Santos. Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de *Software*. INFOCOMP, [S.l.], v. 3, n. 2, p. 8-13, nov. 2004. ISSN 1982-3363. Available at: <http://infocomp.dcc.ufla.br/index.php/INFOCOMP/article/view/68>. Data de acesso: 12 out. 2017.

3. BERNARDO, Paulo Cheque; KON, Fabio. A importância dos Testes Automatizados. Engenharia de *Software* Magazine, v. 1, n. 3, p. 54-57, 2008.

4. Saswat Anand, Edmund K.Burke, Tsong Yueh Chen, John Clarkd, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn. “An orchestrated survey of methodologies for automated *software* test case generation”. Journal of Systems and *Software*, Volume 86, Issue 8, August 2013, Pages 1978-2001.

5. WAZLAWICK, Raul Sidnei “Análise e projeto de sistemas de informação orientados a objetos“ – 2.ed. – Rio de Janeiro: Elsevier, 2011. (Série SBC, Sociedade Brasileira de Computação), p. 30-31.

6. KOSCIANSKI, André; SOARES, Michel dos Santos. Qualidade de *software*. São Paulo: Novatec, v. 3, 2006.

7. MYERS, Glenford J. (2004). The Art of *Software* Testing 2 ed. (Nova Jérsei: John Wiley & Sons), pg 8.

8. PMBOK, GUIDE. Um guia do conhecimento em gerenciamento de projetos. Quarta Edição, 2013.

9. Cruz, Jorge & Jino, Mario & Crespo, Adalberto & Argollo, Miguel. (2006). Suporte automatizado à rastreabilidade em um processo de teste de *software* baseado em documentação.

10. https://cucumber.io/ acessado em outubro de 2017.

11. https://daringfireball.net/projects/markdown/ acessado em outubro de 2017.

12. http://www.devmedia.com.br/como-funciona-a-criptografia-hash-em-java/31139 acessado em outubro de 2017.

13. https://github.com/explore acessado em novembro de 2017.

14. http://www.seleniumhq.org/about/ acessado em novembro de 2017.